Robust Execution of Robot Task-Plans: A Knowledge-based Approach

Örebro Studies in Technology 32

Abdelbaki Bouguerra

Robust Execution of Robot Task-Plans

A Knowledge-based Approach

Title: Robust Execution of Robot Task-Plans:
A Knowledge-based Approach

Publisher: Örebro University 2008
www.publications.oru.se

Editor: Maria Alsbjer
maria.alsbjer@oru.se

Printer: Intellecta DocuSys, V Frölunda 08/2008

ISSN 1650-8580
ISBN 978-91-7668-610-2

Abstract

Abdelbaki Bouguerra (2008): Robust Execution of Robot Task-Plans: A Knowledge-based Approach. Örebro Studies in Technology 32. 175 pp.

Autonomous mobile robots are being developed with the aim of accomplishing complex tasks in different environments, including human habitats as well as less friendly places, such as distant planets and underwater regions. A major challenge faced by such robots is to make sure that their actions are executed correctly and reliably, despite the dynamics and the uncertainty inherent in their working space. This thesis is concerned with the ability of a mobile robot to reliably monitor the execution of its plans and detect failures.

Existing approaches for monitoring the execution of plans rely mainly on checking the explicit effects of plan actions, i.e., effects encoded in the action model. This supposedly means that the effects to monitor are directly observable, but that is not always the case in a real-world environment. In this thesis, we propose to use semantic domain-knowledge to derive and monitor implicit expectations about the effects of actions. For instance, a robot entering a room asserted to be an office should expect to see at least a desk, a chair, and possibly a PC. These expectations are derived from knowledge about the type of the room the robot is entering. If the robot enters a kitchen instead, then it should expect to see an oven, a sink, etc.

The major contributions of this thesis are as follows.

- We define the notion of Semantic Knowledge-based Execution Monitoring *SKEMon*, and we propose a general algorithm for it based on the use of description logics for representing knowledge.
- We develop a probabilistic approach of semantic knowledge-based execution monitoring to take into account uncertainty in both acting and sensing. Specifically, we allow for sensing to be unreliable and for action models to have more than one possible outcome. We also take into consideration uncertainty about the state of the world. This development is essential to the applicability of our technique, since uncertainty is a pervasive feature in robotics.
- We present a general schema to deal with situations where perceptual information relevant to *SKEMon* is missing. The schema includes steps for modeling and generating a course of action to actively collect such information. We describe approaches based on planning and greedy action selection to generate the information-gathering solutions. The thesis also shows how such a schema can be applied to respond to failures occurring before or while an action is executed. The failures we address are ambiguous situations that arise when the robot attempts to anchor symbolic descriptions (relevant to a plan action) in perceptual information.

The work reported in this thesis has been tested and verified using a mobile robot navigating in an indoor environment. In addition, simulation experiments

were conducted to evaluate the performance of *SKEMon* using known metrics. The results show that using semantic knowledge can lead to high performance in monitoring the execution of robot plans.

Keywords: autonomous mobile robots, plan execution and monitoring, semantic knowledge, cognitive robotics.

Acknowledgements

First of all, I would like to express my appreciation to my advisors Lars Karlsson and Alessandro Saffiotti for giving me the opportunity to join the mobile robotics Lab at the center for Applied Autonomous Sensor Systems (AASS) at the university of Örebro. I would also like to thank them for the numerous inspiring discussions we have had during the past five years; their guidance in writing and reading of the drafts of my thesis was extremely helpful. I am also grateful to my opponent and committee members for having accepted to review this thesis.

Many thanks to all the people who have helped me in whatever matter during my stay at AASS as a PhD student; especially, the PhD students at AASS, our lab engineers, senior researchers and secretaries. I am extremely thankful to all persons with whom I have had nice social times outside AASS. In particular, my thanks go to Bourhane, Jay, Elin, Boyko, and the Lilienthals.

Finally, I should mention that this work has been supported by the Swedish KK foundation and the Swedish research council.

Contents

Chapter 1
Introduction

Autonomous mobile robots are being developed with the aim of accomplishing a variety of complex tasks in different environments, including human habitats (e.g., houses, museums, hospitals, etc.) [132, 143] as well as less friendly places, such as outer space including distant planets [105] and underwater regions [59]. To perform their assigned tasks successfully, such robots need to be able to perceive and interact with their environments in an intelligent way. Control architectures are increasingly employing high-level deliberation techniques that allow robots to reason about their actions and resources in an effective and flexible manner. In particular, artificial intelligence planning is used on-board mobile robots to allow them to synthesize their course of action on their own. As a result, robots have the possibility to achieve more tasks without having to be programmed from scratch for each task.

This thesis is concerned with robust execution of robot task-plans in real-world indoor environments. Our main focus is on the ability of a mobile robot to monitor the execution of its plans to make sure they are executed as expected. This ability is essential for the performance as well as the autonomy of the acting robot. That is to say, an autonomous mobile robot needs to be able to detect situations where the execution of its actions diverts from what has been expected to occur.

An autonomous mobile robot must also be able to adapt its behavior in response to unpredictable changes and exceptional situations that might emerge while trying to achieve its tasks. In other words, if the robot is executing a plan to accomplish a certain task, then it is supposed to be able to find alternative ways to continue functioning despite the occurrence of unexpected situations. The thesis presents a general schema to respond to unexpected situations that involve lack of information relevant to execution monitoring. In addition to monitoring the execution of plan actions, we show how such a schema can be applied to respond to failures occurring before or while executing a plan action. The failures we address are due to ambiguity in establishing a connection between symbolic and perceptual data.

1.1 Motivation

In many cases, robots acting in real-world environments face a multitude of challenging issues. Events whose occurrence leads to the disruption of the execution of the robot actions are a typical cause of such issues. For example, a wet floor might cause the wheels of a mobile robot to slip when the robot is trying to navigate to a goal destination. A robot that is executing an action that involves detecting and recognizing objects might find itself not capable of doing that because the lighting conditions are not favorable for taking good pictures of the environment. On the other hand, a robot might not find "the green cup", which is supposed to be on the table in the kitchen, because another robot picked it up and placed it in the cupboard. Another example is of a robot that falls down the stairs because it thought it was navigating in a safer place. The list of examples can be very long, however what should be retained is that in all those cases, the robot failed to execute its actions correctly.

Generally, failures to execute actions are detected when the robot ends up in situations that it did not expect. Such unexpected situations are caused by the presence of uncertainty as well as the dynamics and complexity of the environment. Uncertainty itself might be the result of many factors. The on-board sensors are an important source of uncertainty because they can be unreliable due to noise and physical limitations, such as limited range in case of proximity sensors, and lighting conditions in case of cameras. For instance, navigation failures are mostly caused by errors in localizing the robot within its environment, which is in most cases due to poor sensing (e.g., odometry errors). Unreliable sensors provide uncertain measurements. Those might lead to wrong state estimation (e.g, wrong estimation of the robot pose), which itself might lead to generating wrong controls. As a result, the robot might fail to achieve its goals.

Failures might also be caused by unreliable actuators such as broken motors and flat wheels. Unreliable actuators introduce uncertainty in the outcome of actions, which in turn might result in wrong state estimation; thus, the robot might wrongly believe that it has successfully executed its action when it has failed to do so.

Another source of failures might be the model used by the deliberation and control modules of the mobile robot. A model that does not reflect the true consequences of the robot's actions or the state of the environment can lead to problems that the robot cannot predict. A wrong model would in general result in issuing wrong controls that lead to execution failures. Add to that the potential programming errors and bugs introduced while developing the different modules of the robot architecture [115]. A well-known example of mission failure is the loss of NASA's Mars Polar Lander (MPL) in 1999 when it was about to land on Mars. One possible cause of the failure is speculated to be the software controlling the descent engine; it is claimed that the software shut down the engines because of a false landing signal [13].

Carlson and Murphy [30] collected, during a two-year period, data for studying the reliability of thirteen mobile robots. The authors analyzed the data using some standard manufacturing measures including MTBF (Mean Time Between Failures) and availability[1]. The studied robots included indoor and field robots and came from different manufacturers. The study addressed the failures of robot components that included the control system, effectors, power, sensing, and wireless communication components. The results showed that the field robots failed more often than the indoor robots and that availability was less than 50%. The MTBF was about 8 hours, whereas the probability of failure at a given hour was 5.5%. The components that failed most were the effectors (35%) including the platform itself, followed by the control system (29%) (including the operating system or wired control). The authors included failures caused by humans such as design and interaction failures in a sequel paper [31] where more data was collected (1082 additional usage hours, 75 recorded failures). The MTBF and availability were shown to be improved compared to the first study (MTBF was 3 timers better, while availability reached 54%). The authors argued that the improvements might be attributed to learning from past failures and to repair specialists being better acquainted with the failures. Although the study did not give any clue about executing high-level plans autonomously, it showed that failure is more the norm than the exception. This goes against the claims of Dearden *et al.* [39] and Verma *et al.* [149] that failures are low-probability events.

Taking into consideration all these factors, we can claim that without the ability to monitor the execution of their actions, mobile robots will not notice whether those actions were executed as predicted or failed to produce their desired effects. Responding to the detected unexpected situations, on the other hand, is important because we want the mobile robot to continue acting on its own to achieve its tasks despite the occurrences of contingencies.

Moreover, the ability to detect unexpected situations and respond to them is not only a prerequisite for achieving tasks successfully, but it is also a crucial capacity if we want robots to be efficient and safe in their actions and to their surroundings. In other words, if the robot is able to detect unexpected situations, then it can avoid taking unintentional harmful actions. Efficiency is, on the other hand, the result of not taking unnecessary or counter-productive actions in such unexpected situations.

1.2 Scope of the Thesis

Monitoring approaches of robotic plans have generally focused on comparing the observed effects resulting from the execution of a plan action with its explicit effects, which are specified in an action model; usually, models of actions

[1] MTBF represents the average time to the next failure, while availability represents the ratio of the average time, a robot is completely functional, to the total average time.

used by the planner are used to extract the explicit effects of actions. The aim of the comparison is to establish whether the execution of the action has been successful (i.e., the comparison reveals no difference) or an unexpected situation has occurred. Examples of such approaches include the ROGUE mobile robotic architecture [69] and the work by Fichtner *et al.* [50].

Relying only on using the explicit effects to monitor the execution of plan actions supposedly means that the derived expectations are directly observable. That is, of course, not always realistic in complex environments where checking expectations is inherently a complex process. Therefore, the primary focus of this thesis is on

> *using more advanced forms of reasoning that involve semantic domain-knowledge to derive and monitor implicit expectations related to the correct execution of robots' planned actions.*

By semantic domain-knowledge we mean knowledge about objects and their classes as well as how those objects are related to each other. For instance, in an office environment, an office is a class whose individual instances (objects) are rooms that have at least one desk and a chair; the entities desks and chairs denote themselves classes of pieces of furniture, etc. In the context of monitoring the execution of a robot's actions, semantic domain-knowledge is used as a source of information to logically derive implicit expectations from the explicit ones, i.e., the ones encoded in the action models. The key idea is to compute implicit expectations that can be checked at runtime to make sure that actions are executed as expected. For example, if the mobile robot moves into a room that is asserted to be an office, then it should expect to be in that room (explicit expectation) as well as to see objects that are typical of an office (implicit expectations), e.g., a desk, a chair, and possibly a PC. If the robot is entering a kitchen instead, it should expect to see an oven, a sink, etc.

We also address unexpected situations that occur at execution time. The emphasis of this thesis is on dealing with unexpected situations that are primarily caused by *lack of information* that is necessary for accomplishing robot tasks. We will concentrate on two cases where lack of information is characteristic. First, we consider the case of monitoring the outcomes of an action where the robot has only partial information about whether the implicit expectations hold. Second, we consider perceptual anchoring where the robot tries to identify an object that fits a specific symbolic description and that is relevant to the correct execution of a planned action. More precisely, we are interested in situations where the robot may not have sufficient information to find the correct object due to ambiguity resulting from the robot perceiving more than one candidate object.

In the treatment of the problem addressed in this thesis, we make the following restrictions:

Plan execution: we restrict our work to deal only with the execution of high-level symbolic plans. This means that low-level execution is not addressed in this thesis.

Single mobile robot: we consider the execution of plans of a single mobile robot. Multi-robot plan execution, although challenging, is not addressed in this work.

Indoor environments: the robot acts to achieve tasks in an indoor environment. Tasks that involve outdoor environments are not considered.

1.3 Methodology

We developed our solutions to plan execution monitoring and responding to unexpected situations using standard tools and techniques from the discipline of artificial intelligence. To address the problem of *monitoring the execution of plans*, we employ semantic domain-knowledge as a source of information to compute and check conditions that should hold when an action is executed correctly. We define the notion of *Semantic Knowledge-based Execution Monitoring*, or *SKEMon* for short, and we propose a general algorithm for it based on the use of description logics for representing knowledge. We also develop a second approach of *SKEMon* to take into account probabilistic uncertainty both in acting and sensing. In particular, we allow for sensing to be unreliable, for action models to have more than one possible outcome, and we take into consideration uncertainty about the state of the world. This extension is essential to the applicability of our approach, since uncertainty is a pervasive phenomenon in robotics.

To tackle the issue of *unexpected situations due to lack of information*, we propose to model those situations as a planning problem and employ artificial intelligence sensor-based planning to solve it. As a result, the computed solution takes full advantage of the power of AI planning, i.e., the capacity to reason by looking several steps ahead in order to select the best course of action to solve the problem at hand. Practically, the generated solution is an active information-gathering plan that includes actions to collect runtime information in order to reduce uncertainty about the state of the world.

Since we are dealing with practical problems, the best way to *validate* our proposed approaches is through carrying out an experimental evaluation. To this end, we performed extensive simulation experiments to collect data for the purpose of statistical evaluation of performance. We also implemented our solutions on real mobile robots and ran multiple experiments for different indoor scenarios. Unfortunately, the lack of shared benchmarks in the field makes the evaluation against other solutions impossible. In fact, a common problem that is faced by research works like ours is how to evaluate performance. This problem is mainly due to lack of appropriate evaluation metrics, which are available

in other research areas. It is worth noting that in our work we use standard AI tools that have been validated separately; therefore, our real robot experiments are best seen as test cases that serve as proofs of concept of the proposed approaches.

1.4 Thesis Statement

This thesis is about using standard artificial intelligence knowledge representation and reasoning techniques to achieve a more robust execution of robot plans. The thesis statement is

> *Semantic domain-knowledge and sensor-based planning increase the robustness of autonomous robot architectures because they contribute to the detection and handling of unexpected situations during plan execution.*

1.5 Contributions

The main contributions of the work reported in this thesis are in the area of plan execution in mobile robotics. These contributions are:

- The concept of using semantic domain-knowledge to monitor the execution of robot task-plans. Although the use of semantic knowledge is finding its way in mobile robotic areas, such as mapping and human robot interaction, it is practically inexistent in plan execution. In fact, we are the first to propose to use it systematically to monitor the execution of robot plans, and therefore it is considered to be a major contribution of the current thesis. The contribution is presented in chapter 4 where an algorithm that implements it is also presented. A related contribution includes the development of a probabilistic approach to handle uncertainty in semantic knowledge based execution monitoring. Chapter 5 presents the probabilistic approach and discusses how uncertainty in sensing, action effects, and world stated is taken into account by the monitoring process.

- The study of using sensor-based planning to respond to unexpected situations caused by lack of information. The contribution is formulated as a general schema that models situations of incomplete knowledge as a planning problem. The schema is presented in chapter 6 where it is applied to help the *SKEMon* process to collect information necessary for deducing whether the execution of a plan action have been successful. The same schema is also applied in chapter 7 to resolve situations of ambiguity in finding an object relevant to the successful execution of an action of a task-plan.

Besides the major contributions, the research work leading to the current thesis has resulted in other contributions. These include a probabilistic conditional sensor-based planner (called PC-SHOP) and a hierarchical executor of symbolic conditional plans; both of which are presented in chapter 3.

1.6 Dissertation Map

The reader's guide to the content of the thesis is as follows. In chapter 2, we give an overview of the research topic of the current thesis. We summarize the state of the art of the two subproblems addressed in the thesis, i.e., plan execution monitoring and responding to unexpected situations.

Both formal and practical tools that we used in our research work are presented in chapter 3. We mainly review the THINKINGCAP behavior-based robot control architecture and the deliberative tools we used to implement our solutions. These tools include the sensor-based planners PTLPLAN and PC-SHOP as well as the description logics inference engine LOOM. All of the tools presented in this chapter were already existent, except for the hierarchical planner PC-SHOP and the plan executor that were developed by the author and colleagues.

In chapter 4, we cover our first solution to the problem of monitoring implicit expectations of plan actions. The solution is based on using semantic domain-knowledge to derive and monitor implicit effects of plan actions. This represents a new idea in the field and therefore is considered to be a major contribution of the thesis.

In chapter 5, we go one step further in using semantic domain-knowledge to monitor the execution of robot actions. We take into account quantitative uncertainty in the form of probabilities to model world states, action outcomes, sensing, and the way we interpret expectations in our semantic knowledge.

An information gathering schema is presented in chapter 6 to address situations of lack of information in semantic-knowledge based execution monitoring. The chapter shows how sensor-based planning can be used to generate active information gathering solutions to help in evaluating the outcome of actions.

In chapter 7, we present a solution to recover from a specific type of perceptual failures called anchoring failures. The chapter presents a case study of using the schema of information gathering developed in chapter 6 to handle ambiguous situations in anchoring. These situations arise when the robot cannot identify a perceived object to anchor to a symbol due to uncertainty about properties of perceived candidate objects.

Chapter 8 presents real-robot test scenarios as well as simulation experiments. The real robot scenarios were performed in an indoor environment, and they are intended to show the applicability of the different approaches presented in this thesis. The simulation experiments, on the other hand, are intended for a systematic evaluation of performance.

Chapter 9 presents a summary and a discussion of the contents of the thesis. This chapter identifies the limitations of the proposed solutions and points out possible future research directions.

1.7 Publications

The contents of the current thesis are partially reported in conference and journal papers that are given as follows:

- A. Bouguerra, L. Karlsson, A. Saffiotti. 'Monitoring The Execution of Robot Plans Using Semantic Knowledge'. Journal of Robotics and Autonomous Systems (to appear).
- A. Bouguerra, L. Karlsson, A. Saffiotti. 'Active Execution Monitoring Using Planning and Semantic Knowledge'. ICAPS Workshop on Planning and Plan Execution for Real-World Systems, Providence, Rhode Island, USA, 2007.
- L. Karlsson, A. Bouguerra, M. Broxvall, S Coradeschi, A Saffiotti. 'To Secure an Anchor'. AI Communications', Vol. 21(1), pp 1-14, 2008.
- A. Bouguerra, L. Karlsson, A. Saffiotti. 'Handling Uncertainty in Semantic-Knowledge Based Execution Monitoring'. The IEEE International Conference on Intelligent Robots and Systems (IROS), San Diego, California, USA, 2007.
- A. Bouguerra, L. Karlsson, A. Saffiotti. 'Semantic-Knowledge Based Execution Monitoring for Mobile Robots'. The IEEE International Conference on Robotics and Automation (ICRA), Rome, Italy, 2007.
- A. Bouguerra, L. Karlsson, A. Saffiotti. 'Situation Assessment for Sensor-Based Recovery Planning'. The 17th European Conference on Artificial Intelligence (ECAI), Riva del Garda, Italy, 2006.
- A. Bouguerra. 'A Reactive Approach for Object Finding in Real World Environments'. The 9th International Conference on Intelligent Autonomous Systems (IAS), Tokyo, Japan, 2006.
- A. Bouguerra and L. Karlsson. 'PC-SHOP: A Probabilistic-Conditional Hierarchical Task Planner'. Intelligenza Artificiale, Vol. 2(4): pp 44-50, 2005.
- A. Bouguerra and L. Karlsson. 'Symbolic Probabilistic-Conditional Plans Execution by a Mobile Robot'. IJCAI Workshop on Reasoning with Uncertainty in Robotics (RUR), Edinburgh, Scotland, 2005.

- A. Bouguerra and L. Karlsson. 'Synthesizing Plans for Multiple Domains'. The 6th Symposium on Abstraction, Reformulation, and Approximation (SARA), Lecture Notes in Artificial Intelligence, Vol. 3607, pp. 30-43, 2005.
- A. Bouguerra and L. Karlsson. 'Hierarchical Task Planning under Uncertainty'. The 3rd Italian Workshop on Planning and Scheduling, 9th National Symposium of Associazione Italiana per l'Intelligenza Artificiale, Perugia, Italy, 2004.

Chapter 2
Background and Related Work

Plan execution by mobile robots is arguably a complex and challenging task since it involves dealing with uncertainty and environment dynamics. Autonomy requires that mobile robots be able to detect unexpected situations that might lead to failures to execute their actions. Autonomy requires also that robots try to handle detected unexpected situations on their own in order to successfully accomplish their assigned tasks.

Despite the importance of execution monitoring and responding to unexpected situations in the process of plan execution, it is rare to find literature about research work that is addressed solely to them. Instead, they are usually mentioned briefly when talking about plan execution.

In this chapter, we review research work that has been done in monitoring the execution of plans as well as strategies used to respond to execution failures. Although, our main focus is on plan-controlled mobile robotic architectures, we also give examples of other works that deal with the execution of symbolic plans.

2.1 Monitoring the Execution of Robot Plans

To accomplish their tasks successfully, plan-based mobile robotic architectures need to be able to cope with the issues of uncertainty and the dynamics of the real world that might hinder the correct execution of their task plans. To achieve that objective, plan execution systems employ monitoring techniques and methods in order to make sure that plan actions are executed correctly. The aim of plan execution monitoring is to detect anomalous situations that might lead to execution failure. Thus plan execution monitoring is a fundamental functionality that needs to be implemented in order to achieve robustness in coping with contingencies that might occur at execution-time. Moreover, execution monitoring is a prerequisite for recovering from unexpected situations.

Most plan execution monitoring approaches in mobile robotics use action models to compare the explicit effects of actions to what is observed as a result

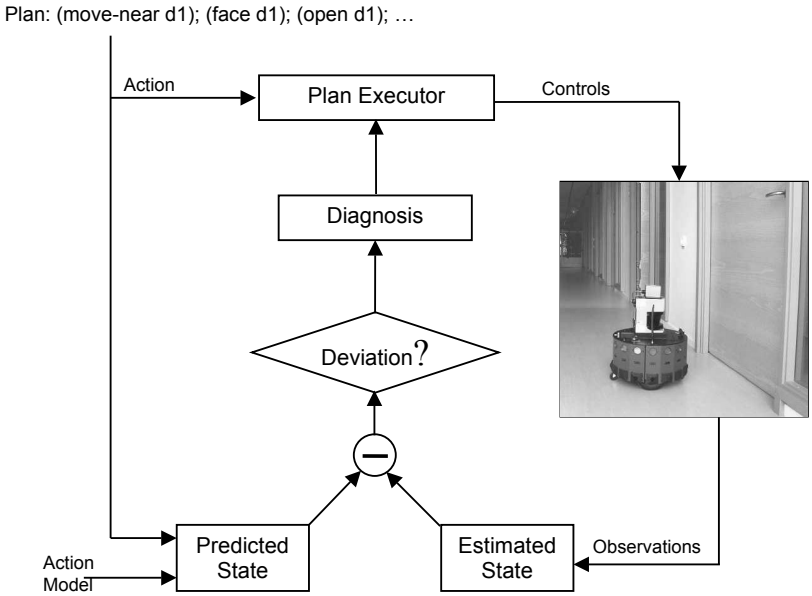


Figure 2.1: Steps of execution and monitoring of symbolic plans by a mobile robot.

of executing those actions (e.g., see the work of Haigh and Veloso [69] and the survey by Pettersson [122]). Other approaches address execution monitoring in an ad hoc fashion, i.e., hard-coded procedures are implemented to monitor specific conditions of interest (see the works of Beetz [6] and McCarthy and Pollack [99]). It is worth mentioning that the terms *nominal* and *expected* are also used to describe the situation that should occur when the action is executed successfully, while the anomalous situation can be qualified as *erroneous*, *faulty*, or simply *unexpected* [147, 50].

Figure 2.1 shows the main steps involved in the execution of symbolic plans by mobile robots. Briefly, the plan executor takes one plan action and translates it into a set of low-level controls, such as velocity and pan-tilt commands. During the execution of the generated controls, the on-board sensing modalities (vision, dead-reckoning, etc.) use the data collected by the robot sensors to compute observations that are used to estimate the actual state of the system. The monitoring module compares the estimated state with the predicted one (i.e., the state that should result after the action is executed correctly). The aim of the comparison is to check whether there is a discrepancy between the two states, i.e., unexpected situations. If a discrepancy is detected, a diagnosis process can be launched in order to identify and classify the occurring unexpected situation. The diagnosis result can be then used by the plan executor to search for a recovery solution.

Example Consider a mobile robot trying to accomplish the task of delivering a mail to a certain office, located in the room identified by the symbol `r1`. The generated task plan could include the following actions to achieve the assigned task:

```
(go-near d1)(face d1)(open d1)(enter r1)(drop-mail r1)
```

The plan includes actions that instruct the robot to move near `d1`, the door of the office, face the door, open the door, and finally drop the mail inside the room. Executing the action `(face d1)` implies that the robot has to orient itself until its front camera is facing the door `d1`. Monitoring the execution of this action relies on the observation of what the robot is seeing to establish the truth value of the predicate `(facing d1)`. If the truth value of the predicate is found to be true, the execution monitoring process deduces that the action has been successfully executed. Otherwise, an unexpected situation is detected, which leads to triggering a recovery procedure with the aim of trying to find a solution, e.g., generating a second plan to achieve the goal of facing door `d1`.

Literature Overview of Execution Monitoring

In the rest of this section, we present an overview of how execution monitoring is addressed in plan-based robotic architectures. For an overview of execution monitoring in other artificial intelligence systems, the reader is referred to the extensive survey by Pettersson [122].

As mentioned above, traditional approaches focus on comparing the estimated state of the world with the one that is predicted to occur when a plan action is executed successfully. In general, the predicted state is computed using predefined models that describe the explicit effects of actions. The first plan controlled mobile robotic architecture Shakey employed the PLANEX system [52] to execute and monitor the execution of plans generated by the STRIPS planner [51]. PLANEX used a data structure called a triangle table where each plan action was annotated with world conditions that made it applicable as well as the predicted effects of that action when executed successfully. The execution of plan actions was carried out by parameterized programs that instructed the robot to perform the desired activity. The representation made it possible to know the preconditions and effects of any portion of the plan. Therefore, the executor would execute only the plan portion that was necessary for accomplishing the assigned task. PLANEX was able to detect whether the so far executed portion of the plan had resulted in its predicted outcome. Moreover, triangle tables allowed the execution monitor to identify situations in which plan actions would no longer be needed to achieve the assigned task.

The LAAS architecture [1] is another plan-based control architecture for mobile robots. Plan execution monitoring is performed by checking the predicted outcomes of executing a plan action with respect to the state computed by the on-board sensing modalities. A failure is raised when a deviation between the two states is detected. The LAAS architecture used the Procedural Reasoning System (PRS) [77, 78] to implement plan execution and monitoring functionalities. In a recent work by Lemai and Ingrand [89], the plan executor of the LAAS architecture was extended to handle the execution of temporal plans. Consequently, other conditions, such as timeouts of action execution, has to be taken into account by the execution monitoring process.

In the integrated planning, executing and learning robotic agent ROGUE [71], hand-coded procedures are used to translate plan actions into sequences of commands that the robot executor understands. The execution monitoring process of ROGUE checks the predicted effects of executed actions using redundant tests. For instance, reaching a specific location is checked both by the navigation system as well as a vision system. Besides plan generation and execution, ROGUE is designed to learn situations where plan execution has already failed. Situation-dependent rules are created accordingly to be used by the planner in order to generate better plans, i.e., plans that try to prevent those failure-inducing situations. For example, ROGUE could learn situations where navigation failed due to busy hours; the on-board planner could then take that fact into account to generate navigation plans that avoided passing through busy locations [70].

The plan executor of NASA's Remote Agent architecture [105] handles both the execution of plan actions as well as execution monitoring. As with the previous architectures, the executor translates actions to be executed into a set of executable steps. The appropriate spacecraft components are then asked to perform the controls necessary to accomplish the executable steps. The execution monitoring relies on information that comes from the model-based Mode Identification and Recovery (MIR) component of Remote Agent. MIR constantly monitors the state of the spacecraft to detect and identify possible component failures. To achieve its tasks, MIR compares information, provided by the on-board sensors, with information generated from the models of the components given the spacecraft's current activities. If the sensor data does not contradict the component models, MIR notifies the plan executor that everything is going as planned. Otherwise an execution failure is reported causing MIR to try to identify the cause of the unexpected situation.

The model-free execution monitoring work reported by Pettersson and colleagues in [121] represents an exception in that it does not use predefined models to predict outcomes of action execution to detect failures. Instead, machine learning techniques are used to learn patterns of failure and success of action execution. The process of plan execution monitoring observes the behavior of the robot and detects whether an action is executed correctly based on what it

has learned. An advantage of this approach is that execution monitoring benefits from past execution experiences to detect execution failures.

Fernández and Simmons [48] use a hierarchy of monitors to monitor the execution of navigation plans. The hierarchy includes monitors that are designed to detect symptoms of specific exceptional situations. For instance, the fact that the robot does not move is considered to be a symptom of the exceptional situation of the robot being stuck. The top-level hierarchy monitors are general monitors designed to cover a large number of exceptions, while the lower-level hierarchy monitors are more specialized and therefore cover fewer exceptions. This means that the more general monitors can be used to detect exceptional situations, which are in general signs of execution failure. The more specialized monitors, on the other hand, can be more informative and thus help in diagnosing the exceptional situation.

In another work by Fernández and colleagues [47], Partially Observable Markov Decision Processes (POMDP) are used to plan for detecting and recovering from execution unexpected situations. POMDPs are a probabilistic formalism that can represent and reason about uncertainty in world state, observations, and results of actions. The authors consider both nominal and exception states for navigation tasks. The actions to plan are those of performing activities that lead to achieving the assigned task as well as actions that collect information about the world state and actions for recovery purposes. Therefore, the computed POMDP policy[1] encloses the execution monitoring process, since faulty states are already identified and taken into consideration when the policy is being generated. To actually determine the resulting belief state at execution time, the execution monitoring process uses the available perceptual information to update the belief state of the robot. Belief update might be costly as a large number of observations and states have to be taken into account. Thus, in a related work by Verma *et al.* [147], Bayesian filters are used to approximate the execution-time belief state. Bayesian filtering techniques are also applied to detect and diagnose exceptional situations caused by hardware faults in planetary rovers [148, 149].

The PRS execution system used by the LAAS architecture is an implementation of the Belief-Desire-Intention (BDI) model of rational agents [24]. In short, a BDI agent organizes its knowledge about the world in a database (Beliefs) that is among other things the result of executing a set of adopted plans (Intentions) to achieve some specific goals (Desires). PRS executes plan actions by instantiating them into a set of predefined declarative procedures that are stored in a library that contains also execution scripts and plans. The process of instantiation takes into account the current state of the world, which is represented in a separate database containing symbolic and numerical facts. This database is continuously updated to reflect the changes detected by the perception system

[1] A policy is a mapping from belief states to actions, where a belief state is a probability distribution over elementary states. In this case, a belief state includes nominal as well as faulty states

of the robot. PRS executes a plan action by trying the different corresponding procedures. If no procedure is applicable, the execution of the action is considered to have failed. It is also possible in PRS to specify monitoring procedures of conditions other than the effects of actions. Thus more complicated monitoring strategies can be defined for each type of action. Other implementations of the BDI model include RAPS [54], Jadex [124], and the commercial system JACK [74]. A common feature of all these systems is that they use hand-coded procedures to monitor events that might affect the execution of the agent actions. Consequently, expectations about the results of actions are explicitly encoded in the monitoring procedure. Thus, to handle new events implies writing new monitoring procedures.

Another execution system inspired by work on intelligent agents is presented by Dias and colleagues in [43]. The high-level plan executor is implemented using the Intelligent Distributed Execution Architecture (IDEA). The basic idea of IDEA is to write control systems as a set of control agents. Each agent uses a model-based reactive planner for reasoning. The proposed architecture was implemented on a planetary rover with two agents; one agent for task planning and another one for executing and monitoring the actions of the task plan.

Other execution monitoring approaches use logic formalisms to describe the dynamics of the environment. An example of a logic-based approach is the work of De Giacomo *et al.* [41] describing a process for monitoring the execution of robot programs written in *Golog*. The working of *Golog* is based on the Situation Calculus, which is a logical formalism for reasoning about the consequences of actions. The execution monitor compares what the robot expects and what it senses to detect discrepancies and recover from them. Discrepancies are assumed to be the result of exogenous actions. The recovery is done through a call to a planner to produce a *Golog* program consisting of a sequence of actions that locally transform the current situation to the one expected by the original program before it failed. The work by Fichtner *et al.* [50] employs the Fluent Calculus, a logical action formalism, to model actions and their effects. Besides detecting discrepancies, the authors describe how such a formalism can be used to provide explanations of why failures occurred, which can be useful to recover from such failures. Lamine and Kabanza propose to use Linear Temporal Logic (LTL) with fuzzy semantics to encode knowledge about successful execution of robot actions [88]. Such knowledge is used by the monitoring process to check the correct execution of the robot actions by considering not only present execution information, but also past one. The monitoring process checks the correct execution of the robot actions by progressing the temporal formulas over the sequences of symbolic states derived from the execution traces. In a related work [80], the authors show that using temporal logic allows to specify monitoring conditions over what should or should not occur in the future as well as past sequences of states with respect to the current state of the world.

There are also approaches that monitor conditions other than the explicit effects of actions. The monitoring approach proposed by Fraser *et al.* [57] considers monitoring plan invariants, i.e., environment conditions that have to hold during the whole execution episode of a plan. The Rationale-Based Monitoring approach [146, 99] and Propice-Plan [42] monitor features of the environment that can affect the plan under construction. When a feature is detected to be a potential threat to the execution of the plan, the planning process takes into account such information and adapts the plan under construction accordingly.

The assumptive mobile robotic architecture by Nourbakhsh and Genesereth [113] focuses on interleaving planning and plan execution to cope with uncertainty due to lack of information through the use of assumptions to simplify the planning task. For example, when there are several hypotheses about the location of the robot, a simplifying assumption can be to consider that the robot is in the most likely location. Because the planning assumptions might turn out to be wrong, the execution monitoring process must continuously check that they are not violated. This ensures that the robot does not execute actions that might result in disastrous outcomes.

Beetz proposes to use structured reactive controllers (SRCs) to implement the execution and monitoring system of precomputed plans for office-delivery tasks [5]. SRCs are collections of procedures intended to be implementations of reactive controllers that run concurrently. The SRCs include two types of plans. The first type is called structured reactive plans; they are used to specify the actions needed to achieve user requests. As in the assumptive planning architecture above, the structured reactive plans can be created based on assumptions about some features of the environment, e.g., doors of offices to deliver mail to are all open. The second type of plans are called policies; they are in charge of maintaining conditions that are necessary to the execution of the first type of plans. They are also used to monitor the execution of the structured reactive plans. Policies to monitor assumptions, made by the first type of plans, need to be specified explicitly.

Ontological control [12] was proposed to monitor the execution of sequences of actions used to control industrial plants. The main focus of ontological control is to detect deviations of the model-based expected behavior of the controlled system and classify those deviations according to what caused them. First, the deviations can be caused by external actions, which might result in disturbing the functioning of the controlled system. Second, deviations might be caused by violations of ontological assumptions representing expectations that are due to faulty action models. These violations are deduced based on the assumption that the actuators are reliable, i.e., the execution of an action gives always the same actual outcome, but the model does not reflect that outcome.

2.2 Responding to Unexpected Situations

To continue acting autonomously, mobile robots need to be able to adapt their behaviors in response to the detection of unexpected situations while they are executing their plans. That is to say, an autonomous mobile robot is supposed to be able to find alternative ways to continue acting, in case the execution of its actions do not succeed. As it is mentioned by Turner *et al.* in [144], handling unexpected events is a difficult task because in many cases they are hard to detect. Moreover, it is even harder to identify their causes and their severity, which makes it difficult to decide how to cope with the problems they cause.

Bjäreland observes that recovery from execution failures is a function that is difficult to characterize because of the different interpretations associated with it [11]. However, much of the research carried out in autonomous mobile robotics views execution recovery as part of the system in charge of plan execution or as a process that uses the functionalities of such a system. For instance, in [11], [41] and [50] both detecting unexpected situations and responding to them are constituent parts of the execution monitoring process.

In this section, we survey the different approaches and strategies used to recover from execution failures. The primary focus will be on recovering from failures of executing high-level task-plans. However, this does not prevent us from citing references related to recovery strategies at different levels of execution.

2.2.1 Response Strategies

Upon the detection of an unexpected situation such as an action execution failure, the recovery mechanism has to take an immediate action to allow the robot continue its course of action; if it is not possible to do so, the recovery mechanism should ensure that the robot is put in a safe state. Recovering from plan execution failures can be done in different ways. In systems that support backtracking at execution time, recovery might be to backtrack to a working state, which is similar to rolling back in software systems such as database management systems. However, this technique cannot be used solely in mobile robotics, simply because there might be no possibility to backtrack to a working state. Therefore, engaging in the computation of a correction procedure is necessary in such situations. Another way is to identify potential failures in advance and compute recovery procedures to deal with them. Such procedures are executed at runtime, whenever failures associated with them are detected.

Data about past failures and how they were recovered from can be used to prevent the occurrence of failures. There are two ways to do so. First, the failure and the procedure used to recover from it can be classified and stored. Second, the available data about failures can be used to improve the model of action and world to avoid subsequent similar situations. In our survey, we classify strategies for handling unexpected situations of plan execution according to

when those strategies compute the response procedures, i.e., before launching the execution of plans (off-line) or at plan execution-time (on-line).

2.2.2 On-line Response Strategies

Here, the robot engages in taking a course of action to achieve a certain task and postpones the computation of responses to recover from failures until they occur at execution-time. On-line recovery strategies include replanning, plan adaptation, and reconfiguration of functional modules. Clearly, to be able to compute recovery solutions at execution-time, mobile robots have to be endowed with situation assessment capabilities to help them identify what went wrong and possibly why. As observed by Fernández and Simmons, accurate situation assessment is of primary importance for computing correct recovery solutions [48]. Not only do correct recovery solutions help to recover from execution failures, but they also contribute to avoiding the occurrence of new failures. In other words, if the recovery solution is not correct, then its execution can lead to other unexpected situations.

Replanning

Replanning is a technique widely used within plan-based control architectures in mobile robotics. The use of replanning dates back to the early days of mobile robots where it was used within PLANEX, the plan executor on-board Shakey the robot [53]. Triangle tables were used within PLANEX to reuse a plan if one of its actions failed to execute [52]. If no plan portion could be executed, the planning engine STRIPS [51] was invoked to compute a new plan to reach the original goal from the current state. Replanning is used within several other mobile robotic architectures including ROGUE [71], the navigation architecture ThinkingCap [134], and NASA's remote agent [105].

Responding to unexpected situations using replanning presupposes that the robot is executing a sequence of actions (plan) to reach a goal state. Every time a plan action is executed, its effects are checked by the execution monitoring process so that unexpected situations can be detected. The preconditions of the next action to execute are also checked by the execution monitoring process to determine if the action is executable in the current state of the world. Whenever an unexpected situation is detected, the execution of the current plan is suspended and a recovery procedure is launched. Computing a recovery solution involves calling the task planner to find a new plan that transforms the current (faulty) state into the goal-state of the failed plan. Since the goal-state might have some facts achieved by the so-far executed actions, the goal-state of the recovery task might be considered to be the set of unachieved facts. This idea is used in the temporal planning and executor system IxTeT-eXeC [89].

When the planner finds a new plan, the executor schedules it for execution, otherwise a permanent execution failure is declared leading to the cancellation

1. **Put the robot in a safe state**
2. **Compute the current state**
3. **Find a plan to reach the original goal**
4. **If plan found, execute it,**
5. **Else declare** *permanent failure*

Figure 2.2: Main steps of a replanning recovery-strategy.

of the current task. An abstract replanning schema is given in figure 2.2. The first step in the abstract schema is optional, and depends on the severity of the faulty state. Some architectures such as ROGUE allow the robot to continue executing other tasks meanwhile planning to solve other problems [69]. A typical scenario where replanning can be used is a robot navigating in an indoor environment. If the current route of the robot is blocked, the planner is called to try to find another path leading to the goal location.

Replanning is a straightforward recovery strategy, since recovery can be considered as another planning problem with the current "faulty" state as the initial state while the goal-state to reach is the same as that of the failing plan. However, the efficiency of replanning as a recovery strategy depends to a great extent on a good state estimator and a good action model. The role of the state estimator is important for replanning because the plan generated depends on the initial state of the planning problem. If the initial state does not reflect the state of the world, then the generated plan might be non-executable. Even if it is executable, it might not lead to the desired goal-state. On the other hand, having a good action model is important for creating a plan that predicts as closely as possible the actual outcomes of the actions when they are executed in the real world.

Plan Adaptation

Plan adaptation is another strategy that is used to cope with unexpected situations at the time they occur. The key idea of plan adaptation is to keep the current plan in execution and try to correct the portion of the plan that has failed, while unaffected sub-plans continue to be executed. This strategy is adopted by architectures that execute partial-order plans such as in Cypress [153, 152] where it is called asynchronous run-time replanning, and in IxTeT-eXeC [89] where it is referred to simply as plan repair.

When correcting portions of a plan while continuing executing others, certain issues arise that have to be addressed by the system. The first issue arises when the state of the world, resulting from the execution of a non failed plan portion, affects the predictions of the portion being repaired. Consequently, the planner/executor has to envisage how to integrate the replanned activity

with the rest of the executing ones. The second issue concerns making sure that the new replanned portion does not invalidate the other sub-plans, i.e., ensure conflict-free sub-plans. As stated by Pell *et al.* in [120], resolving one problem can lead to new problems, hence not only does the recovery procedure have to make local repair, but it also has to take into account the overall constraints related to correct execution.

Both Cypress and IxTeT-eXeC call the same planner that generated the initial task-plan to perform a search for a local repair plan. An example of recovery using local plan repair is described by Lemai and Ingrand in [89] where a mobile robot is asked to carry two objects to their destination locations. As the robot is executing its plan to achieve the task, execution failure occurs when the first object is accidentally dropped on the floor. The robot continues executing the portion of the plan related to the second object, while the portion related to the first object is being repaired by adding actions to pick up and carry the fallen object.

Beetz presents another plan adaptation framework where predefined processes are embedded in the task plans with the aim of repairing them, should a belief change be detected [6]. Those processes perform execution-time plan adaptation in two stages: first a reactive response is produced, then a more deliberative response is performed to revise the currently executed plan. Belief change is detected either when failures occur or when opportunities arise. Plan adaptation is specified by a set of specialized methods that are defined using a set of transformation rules that might even invoke a planner.

Other architectures that support runtime plan adaptation include the two-layered robot programming framework CLARAty [111] that employs in its decision layer the planning system CASPER [33]. In CLARAty, the planner is continuously in interaction with the executor. In other words, in each cycle CASPER is called to compute the effects of updates to the current state and goals on the current plan. If unexpected situations are detected, the planner tries to repair the currently executed plan.

It is worth mentioning that, from a theoretical point of view, trying to modify a plan by keeping as much as possible of the old plan (failed plan) can be harder than planning from scratch [110].

Reconfiguration

One way to provide reliable acting of mobile robots is to make them fault-tolerant [49, 93], i.e., despite the presence of faults, they can continue acting [96]. One of the possible strategies to fault-tolerance is the use of redundant modules (software or hardware) where recovery is performed by reconfiguring non failing modules to compensate for the failed module.

Among the robotic architectures that use alternative procedures to achieve tasks, we cite the three-layered architectures ATLANTIS [62], REFLECS [65], and the executor PRS-CL[106]. ATLANTIS comprises three components: a re-

active controller, a deliberator and an executor. It is the responsibility of the executor layer to decompose higher-level tasks into low-level tasks and sequence the primitive activities (reactive sensorimotor processes) achieving them. It also keeps a set of methods for each task. If the execution of a task fails, an alternative method is tried instead. PRS-CL can be considered as an executor that provides the functionality of the execution layer in ATLANTIS. PRS-CL achieves tasks by a set of predefined procedures referred to as *Acts* depending on the observed state of the world. REFLECS, on the other hand, addresses behavioral cycles that manifest themselves in mobile robots' schema-based reactive architectures, where control is specified as a configuration of modes (*on*, or *off* mode) of schemas. REFLECS incorporates a deliberative module that monitors for failures resulting from repeating behaviors due to local optima. The response consists in computing a new configuration of schema modes using predefined methods. Typically, the methods determine changeable schemas and tasks that can replace them. Schemas to be changed are then disabled by setting their mode to *off* while schemas replacing them get activated by setting their mode to *on*. In [123], Pirjanian presents a formal description of a voting scheme that shows how redundant behaviors can be combined to reach a more reliable execution than when just one behavior is used.

The plan executor of NASA's Remote Agent architecture [120, 105] includes two sub-modules: EXEC and MIR. EXEC is a reactive plan execution system that provides control procedures, task decomposition and scheduling as well as concurrency. The functioning of EXEC is based on the RAPS [54] procedural language, which is used to define redundant methods to achieve tasks. MIR, on the other hand, is a deductive model-based mode identification and reconfiguration system. It is used to determine the current state of the spacecraft and to recompute the configurations of hardware components. MIR is also called by EXEC to compute sequences of actions to restore function and to recover from execution failures caused by components of the spacecraft. For instance, if the action of staring an engine fails because of a stuck valve, MIR can generate a sequence of actions (such as opening and closing valves) to reconfigure the components that would make it possible to start the engine.

The SFX-EH architecture [104] addresses the detection, classification, and recovery from sensing failures. In addition to recalibration and corrective actions, reconfiguration is one of the strategies used to recover from sensing failures in SFX-EH. Reconfiguration relies on the presence of redundant logical sensors, which are perceptual processes that can be used by the same perception schema of a reactive behavior. If a perception schema detects that one of its logical sensors has failed, then a new configuration of the other logical sensors is generated to compensate for the failing one. If no reconfiguration can be generated, the corresponding behavior is deactivated and possibly a new one that uses a different perception schema is activated.

Reconfiguration may also be used to restore a robot's functionality following a physical damage by adjusting the parameters of the robot's controller.

Bongard and Lipson [18, 19] propose an evolutionary strategy to estimate, in a first stage, damage hypotheses incured by a simulated legged robot when walking forward. In a second stage, the controller is evolved to cope with the physical damage. The simulated legged robots are controlled by a neural network. Upon the detection of a failure (e.g., due to a broken leg), the first stage of the evolutionary algorithm is used to estimate a damage hypothesis on the basis of a limited number of predefined damage causes. The damage hypothesis and the controller of the physical robot are then fed to the second stage of the evolutionary algorithm. The aim is to evolve the controller through the generation using a simulator of the physical robot for fitness evaluation. Then, the evolved controller is downloaded to the robot for testing. Sensor data resulting from the test is used by the estimation stage together with the evolved controller to evolve the robot's simulator so it better reflects the physical robot.

2.2.3 Off-Line Response Strategies

Off-line strategies imply the computation of the response to cope with execution failures before the robot starts the execution of its tasks. In this class of responses, the exceptional situations are anticipated, possibly because the robot has already encountered them, or they are identified during the design phase. Off-line strategies include contingency planning as well as precomputed failure-response procedures and plans.

Contingency Planning

As outlined before, uncertainty in sensing as well as in the state of the world and in the outcomes of actions represent a cause of failure of plan execution. One way to cope with execution failures is to reason about uncertainty when plans are generated, i.e., generate contingency plans. The key idea of contingency planning is to plan in advance for potential contingencies by explicitly encoding responses to possible failures as plan branches of the main plan. A lot of research work has been carried out to address the issues of uncertainty and contingencies in planning. For an overview of techniques of planning under uncertainty, the reader is referred to the survey by Blythe [16] and to the recent book by Ghallab *et al.* [63] about artificial intelligence planning.

Using contingency planning to handle failures involves the definition of a set of actions that collect information at execution-time, so it is possible to determine the course of action to follow. The main issue with contingency planning is that the size of the plan increases exponentially with the number of contingencies. Thus, some techniques aim at planning only for contingencies judged as to have a severe impact on the execution of the main plan (plan without contingencies).

In the following, we review three planning systems aiming at selecting contingencies to handle failures that might affect the overall value of the plan. The

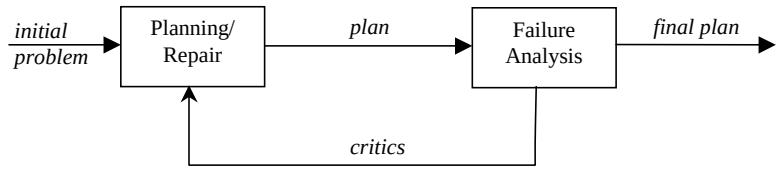


Figure 2.3: The two stages of incremental planning under uncertainty.

three approaches share the same incremental structure. They start by building an initial plan, which is subsequently analyzed for failure places to determine where it is best to insert plan branches to deal with failures. This process of planning and plan analysis continues until producing a plan with a desired value expressed either as a success probability or expected utility. Figure 2.3 schematizes the two-stage incremental process. It is worth noting that all three systems do not consider the cost (or value) of replanning in their contingency selection.

The first planning system we review is WEAVER [14, 15], which is an probabilistic planner that takes into account actions as well as external events that can change the state of the world. The external events have a probability of occurrence conditioned by the satisfaction of some conditions in the state of the world. The focus of WEAVER is on generating plans that solve the planning problem with a certain degree of success (expressed as a probability). It relies on a generic planner to find plans, and then tries to correct them in order to reduce the effects of external events that might take place at execution time. The planner does not consider external events when solving the planning problems. Instead, they are introduced by a failure analysis module that translates the plan into a Bayesian belief network. The objective of failure analysis is to calculate the success probability of the plan and to look for events whose occurrences can lead to failures. The result of the analysis module is used to introduce corrective actions that either undo the effects of the external events, negate their preconditions, or reduce the occurrence time of an event. The same process repeats with the corrected plan until producing a plan satisfying the success probability.

MAHINUR [117] is also an incremental planner that addresses the question of which plan-execution failures should be planned for. MAHINUR starts also by building a plan with a non-zero probability of success. Then, actions are added to take into account the failure of the plan branch whose utility is maximal. The planning process stops if the utility of the resulting plan exceeds a preset threshold or there is no time for extra planning. The identification of the contingency with the greatest impact on the utility of the overall plan is done by selecting the contingency with maximum disutility.

More recently, the Just In Case planning system, JIC, [40, 25] was proposed to cope with uncertainty in action duration and continuous resources for space rovers. The JIC planner starts by generating a first version of the plan (called seed plan) taking into account constraints over resources such as their availability and how they are used by the plan actions. The generated seed plan is constructed to achieve the rover mission goals with maximum expected utility. Next, the seed plan is checked for possible failure places to determine the best point to insert a contingency plan so that the overall utility of the plan is maximized. The procedure of selecting the contingency branch with maximal utility would involve calculating the utility of the branch; a calculation that cannot be done exactly unless the plan of the branch is fully constructed. Therefore, the authors propose to approximate the utility of a branch without searching the actual planning. Instead, a reachability graph that reaches the goals is constructed and then the utility is back-propagated in the graph. Besides the different types of contingencies they consider, the JIC planner and Weaver differ also in the criterion used to generate plans. Weaver estimates success probability while the JIC planner estimates utilities.

Although, the three systems share their main incremental structure, there are differences between the way they consider failure points. Weaver relies on improving the probability of success, and therefore branches are added where it is estimated that the probability of success will be improved. In Mahinur, and the JIC planner, improving the overall utility is used to repair the plan, which according to [40], makes it possible plan branches in the right place compared to what Weaver does. On the other hand, Mahinur differs from the JIC planner in the way the utilities of the contingencies are estimated. Moreover, the JIC planner considers continuous time and resources, while Mahinur does not.

Universal Planning

Another approach to deal with execution failures is to take into account all situations (i.e., nominal and unexpected situations) that might arise at execution-time and associate a reaction for every possible situation. This idea was proposed by Shoppers in [136] to generate universal plans to guide robots in achieving their goals. It is worth mentioning that the idea of universal planning has encountered critics regarding the exponential size of all the potential situations that an agent might encounter at execution time [64].

An example of formalisms that generate plans for all possible situations is Markov Decision Processes, or MDPs for short [125]. MDPs have been widely used within mobile robotic architectures as robust control formalisms, e.g., [29, 66]. MDPs are a formal model that is used to generate policies (universal plans) taking into consideration costs and rewards as well as uncertainty about action outcomes. They generate policies that maximize the expected total return resulting from the application of actions in states [23, 90]. The use of an MDP formulation to solve the robot tasks makes it possible to model faulty states as

possible results of actions, and therefore associate costs with ending up in such states. As a result, the generated optimal policy takes into account faulty states and associates actions with them in order to recover to more desirable states.

Partially Observable MDPs [81], or POMDPs for short, extend MDPs to model environments with partial observability, i.e., where there is uncertainty about the actual state of the world. Thanks to their ability to represent explicitly different forms of uncertainty, especially about sensing and acting, POMDPs have been reported as successful tools in controlling mobile robot to achieve indoor navigation tasks [114, 139]. The work by Fernandez and colleagues [47] is an example of using POMDPs in an architecture dedicated to the supervision of indoor mobile robot navigation. A supervision policy is computed off-line, and the process of identifying unexpected situations and recovery from them is performed on-line. The POMDP model classifies states as nominal states and exceptional states (states to recover from). The exceptional states are determined as a combination of known exceptional situations (e.g., non navigable path, perception problems, etc.). Actions can be either recovery actions or actions to accomplish a certain task such as navigating to a specific location. An example of a recovery action is to call the path planner to find an alternative path, when the robot believes it cannot follow a corridor because it is blocked.

Despite the attraction of using POMDPs due to their ability to handle many aspects of uncertainty, their use is always constrained by the size of the problem they try to solve. POMDPs become intractable for problems with just hundreds of states and few observations even when approximate solutions are used [145]. Nevertheless, there are techniques that can give approximate solutions for certain classes of problems with far greater state spaces [128].

Precoded Failure Responses

Response strategies under this category involve off-line coding of procedures that the plan executor has to try in order to cope with known failures. This is equivalent to organizing failures and their responses in a look-up table. At execution time, the execution monitor is active to detect failures. Upon the detection of a failure, a process of failure identification is launched to identify the encountered failure. The identified failure can be used to extract the corresponding response procedure from the library of responses [48].

Precomputed failure strategies can be made more reliable by adding a debugging component to assess the influence of recovery procedures on the occurrence of subsequent failures. One way to do that is to debug execution traces of plans to search for dependencies between recovery actions and failures. Dependencies are analyzed to build possible hypotheses of caused failures. Identifying the most likely hypothesis helps modify recovery actions and add new ones [76, 75].

Precoded responses are used within the Procedural Reasoning System [78] and its derivatives PRS-CL [152], and PRS-Lite [106], where failures are re-

sponded to through the execution of predefined declarative plans/procedures. A procedure is described by a body specifying the steps to follow to recover from the failure. Similarly, the RAPs (Reactive Action Packages) execution system [54] refines abstract plan actions into detailed instructions at execution-time by choosing a method for the next step in execution from a preexisting library. Methods to execute abstract actions are selected depending on the context of the run-time situation, meaning that at run-time the situation of the world is determined to index the appropriate method in the RAP.

The Task Control Architecture (TCA) [138] is also an execution framework that has been employed to control several mobile robots. It supports the definition of hierarchical exception handling procedures that are associated with specific task nodes in the tree of tasks to be executed. If an exception is detected while executing a task node, the associated procedure is called to handle the exception, and hence the possibility of changing the task tree itself. In case the exception handler cannot recover from the failure, it is possible to a call another procedure higher up in the tree to try to handle the exception.

Clearly, using precoded procedures is limited to working with known failures. As mobile robots work in highly dynamic environments, relying solely on this approach to achieve reliable execution would not work in face of unknown and unanticipated failures. For this reason, some of the mobile robotic architectures employ this strategy together with other strategies, such as plan adaptation, reconfiguration, and/or replanning [153]. Moreover, using precomputed responses to cope with failures implies using robust failure identification and diagnosis so that the best recovery procedures are selected. The autonomous underwater vehicle controller ORCA [144] focuses most of its efforts on identifying unanticipated events and the assessment of their importance using a backward chaining fuzzy rule-based diagnosis system. A response is selected for an event if it is assessed as important.

2.2.4 Failure Prevention

In this section, we give an overview of work addressing learning from past experiences and from interaction with the environment to improve the environment model and/or the action model used by planning systems. The aim of such work is to make it possible for the used planning system to generate plans that improve execution reliability and avoid failure. Learning methods involving supervision or methods that learn models completely from scratch [119] are not considered here, since we assume that the robot has already a sound and complete planning system and needs only to tune its models to cope with uncertainty about the environment and the outcome of actions. One typical example of learning systems is OBSERVER, [150], which relies on learning by observing a set of execution traces generated by an expert. The execution traces are used to learn operators that are subsequently refined by solving practice problems.

The TRAIL [8] learning system learns from execution traces generated by an expert, but it is more interesting since it learns from its own experience as well. TRAIL learns domain-specific action models to be used by a reactive agent to generate plans to achieve a certain goal. The action model uses teleo-operators, TOPs for short, which are processes used to maintain the execution of a primitive behavior until a condition becomes true. TRIAL aims to learn preconditions of TOPs that have to be maintained until the realization of the intended postcondition. Learning is carried out after the execution of a plan action fails. Thus learning is interleaved with execution of actions. TRAIL learns preconditions of TOPs by refining them such that conditions that precede the successful execution of a TOP are preserved and those that precede a failing TOP are excluded.

Schmill and colleagues [135] present an example of a learning system that learns to improve a probabilistic action model. The learning system learns both the preconditions and the effects of actions. The effects of an action are extracted by partitioning experiences (expressed as multivariate time series containing sensor measurements collected while executing an action) into clusters according to a similarity metric measure. The resulting clusters are considered to be the different outcomes of the action involved in the different analyzed experiences. Learning the conditions that can be used to predict the outcome of actions is an induction process that uses sensor data that precede the execution of the target actions. The induction process uses decision trees to learn from the initial conditions for each action. In the constructed trees, leaves represent the clusters while non-leaf nodes represent precursor conditions expressed as sensor data. Conditions are formed as a disjunct of conjunctions over the non-leaf nodes.

Another strategy to help robots learn from their previous experiences is to build control rules to be used by the planner in order to avoid situations that would make the execution of the plan fail. The ROGUE robotic system [71, 69] employs a learning module to do just that. The learning module is used to enrich the planning domain with control knowledge expressed as a set of rules extracted from the execution traces of plans. The aim of the learning process is to identify situations where actions could not be executed as predicted. Success and failure outcomes of actions are considered to be learning opportunities that trigger the extraction of situation features related to learning events. Features can be high-level such as speed, and time of the day. Execution-level features such as sensor readings, current location, etc., are also considered. The learning system associates a cost with situational features and learning opportunities. Regression trees are used to create rules aimed at controlling planning search. The control rules, then, can be used to select goals or reject them based on their cost, i.e., if the cost is high then the goal is rejected, and if it is low then it can be accepted [70].

ERA [4] is another approach that uses regression trees to build models of actions. As in ROGUE, ERA builds regression trees applied to data collected by

the robot while executing its actions in the environment. The collected data includes a description of the state of the world before executing an action, the action taken in the world and the resulting state. The regression trees are used to predict the expected errors in the outcomes of actions under specific environment condition. The action models can then be used with a path planner to determine the influence of terrain conditions on navigation and to predict the distance the robot would travel within different regions.

Planning, execution, and learning have also been integrated in a system called LOPE [98]. The system generates plans, and then proceeds to their execution. While executing actions the system learns a model of its environment by observing the effects of its actions on the environment. Observing the effects of actions allows the system to create operators if they do not exist in the knowledge database, or adapt them to the changes in the environment through the use of generalization heuristics and a reinforcement strategy that gives more credit to operators with more execution successes and punishes operators that have failed often.

McDermott and Beetz [7] present an approach that does not rely on learning. Instead, they propose to debug plans while they are executed to prevent probable execution failures by projecting the effects of plans at execution time. The idea is to compute a set of sample execution scenarios to help predict subsequent failures that might affect the quality of the plan. Detected failures are used as indexes to retrieve precomputed transformation rules to be applied in order to improve the plan under execution. Following the same idea of preventing failures, Py and Ingrand [126] propose an execution control component called Request & Resource Checker (R^2C) whose aim is to verify that requests for functional modules would not result in an inconsistent state. The R^2C component relies on a the definition of constraints that specify the acceptable and unacceptable states of the different components forming the functional level of the robot. The role of R^2C is to maintain a derived formula true in all the states of the system. Should a request result in an inconsistent state, R^2C responds with an action to avoid it, e.g., rejecting the request.

2.3 Conclusion and Discussion

The ability to detect and respond to unexpected situations is essential for systems designed to act autonomously in real-world environments. We believe that detecting and handling unexpected situations constitutes the core of autonomy, simply because an agent that cannot handle and reason about unpredicted states on its own is doomed to fail to achieve its tasks where external help is not available.

The first part of this chapter was devoted to reviewing research work dealing with plan execution monitoring. We have seen that mobile robotic architectures usually use two types of execution monitoring approaches. First, there are approaches that rely on hand-coded procedures to monitor and detect un-

expected situations. This class includes mainly reactive executors such as PRS, RAPS, and TCL. What characterizes theses approaches is the lack of flexibility in detecting unspecified execution failures. In fact, one needs to write new monitoring procedures, whenever new conditions are to be monitored. The second class of monitoring approaches uses models of plan actions to determine what results to expect when a plan action is executed successfully. Among the architectures using such approaches, we find PLANEX, ROGUE, and the LAAS architecture. We have also seen that there are monitoring approaches that use logic formalisms to code knowledge about the dynamics of the world with an emphasis on the results of actions taken by a robot.

In the second part of the chapter, we reviewed research work aiming at responding to situations that were not expected to occur at execution time. We classified strategies designed to deal with such situations into two classes: off-line and on-line strategies. On-line strategies compute response procedures at execution time, i.e., after the robot has detected that the execution of an action has failed. Computing response procedures on-line requires that the robot be able to reason on its own to find the best sequence of actions that would help in achieving the desired effects. We can identify two main approaches to recover from failures at execution time. First, there are strategies that rely on Artificial Intelligence planning techniques either to plan from scratch to achieve the goals of the failed task or to adapt the current plan to the current situation. Second, other strategies borrow ideas from the fault-tolerance community to use redundancy as a means to handle failures. The key idea is to equip the robot with redundant modules (hardware or software) that can provide the same service. In case of a failure of one module, the system is reconfigured to bring the unexpected state to a working state.

The second class of recovery strategies encompasses techniques that compute recovery procedures off-line, i.e., before the robot starts the execution of its actions. The key idea behind off-line strategies is the computation of anticipatory counteraction of predictable failures. One way to do that is through making the task plan incorporate branches that handle potential contingencies that might arise at execution time. Another approach is to prepare a universal plan where all possible situations (nominal as well as faulty ones) have actions associated with them. The latter approach has been criticized regarding its realization in real environments. We also reviewed strategies that do not use planning where experts provide their knowledge to handle predictable failures and contingencies as hand-coded procedures.

AI learning techniques have been used to prevent failures by learning rules from previous situations that led to failure. The rules can then be used by the task planner to avoid taking decisions that might degrade execution or lead to failure. As the actions of a robot might lead to failures, learning has also been employed to learn better action models that reflect more accurately the true effects of actions. As a result, the robot can predict better the outcomes of its actions.

We believe that none of the reviewed strategies is the absolute solution to respond to unexpected situations. Instead, each strategy has its strengths and weaknesses. On-line recovery strategies might be the ultimate solution if they are given enough time and a sufficiently accurate model of the environment. However, there are situations where the environment does not wait for the robot to make plans and deliberate about its actions. In addition to that, models are only available for a subset of the possible interactions with the environment. Off-line recovery strategies on the other hand, have the advantage of being validated and tested thoroughly before they are loaded on-board. However, the designers can anticipate only some of the faulty situations, making it difficult to handle unexpected ones.

Between the two extremes of computing failure responses, i.e., off-line versus on-line, the three tiered architectures, such as Cypress [153], Rogue [71], the Remote Agent [120], and IxTeT-eXeC [89], provide a middle ground where the executor executes the actions provided by the planner using precomputed procedures. The execution procedures are programmed to handle known failures at execution time while the planner is used to find an alternative plan if failures cannot be recovered from by the executor.

Regarding how this thesis relates to the presented material in this chapter, we have two main remarks. First, existing approaches for monitoring plan execution have their primary focus on checking whether the explicit expectations of actions are verified in the estimated state of the world. This relies on the assumption that the effects to monitor are directly observable. However, in real-world environments checking expectations can be a complex process. In this thesis, we focus on presenting a knowledge-based approach to derive and monitor *implicit* expectations, i.e., expectations that are not given in the action model. To our knowledge, we are the first to propose such an idea.

Second, we can also claim that there is lack of thorough research work devoted to studying and handling unexpected situations of plan execution. In fact, responding to unexpected situations is generally addressed as a secondary functionality of plan execution. Moreover, plan execution systems tend to respond to unexpected situations in the same way, i.e., by using the same response strategy. We believe that different types of unexpected situations need to be studied separately. In this thesis, our focus is on unexpected situations that are due to lack of information. Therefore, we are able to provide a strategy tailored to deal with such situations in an effective way through using sensor-based planning.

Chapter 3
Tools

This chapter covers the tools we used to implement our solutions to the problems addressed by the current thesis. We start by giving an overview of sensor-based planning and the two planning tools PTLplan and PC-Shop that we employed to implement active information gathering solutions to deal with situations characterized by lack of information. Then, we take a look at description logics and the Loom system used to represent and reason about semantic domain-knowledge. Next, the ThinkingCap robot control architecture is described together with how the execution of symbolic conditional plans is carried out. At the end of the chapter, we present our Magellan Pro mobile robots that have been used to conduct the different real-world test scenarios.

It is worth noting that both PC-Shop and the symbolic plan executor described in this chapter have been created as part of the research work leading to this thesis. The other tools were developed in the context of other research work, and therefore more information about them can be found the papers and articles referencing them.

3.1 Sensor-based Planning

Artificial intelligence planning is a problem-solving paradigm that is used to find a course of action to reach a specific goal state starting from an initial state. Classical planning approaches, such as STRIPS [51] and Shop [107], rely on simplifying assumptions to generate plans. One important assumption they make is that the state of the world is changed only by the execution of a plan action and that effects of actions are deterministic. They also assume that the agent executing the plan will always have complete information about the state of the world. Thus, those approaches are considered to be "open-loop" since they do not use feedback from the environment to determine what action to execute next.

Classical planners would in most cases be unable to handle real-world problems, because of the simplifying assumptions they make. In a real-world situ-

ation, planners do not have immediate access to all the relevant information. Moreover, the presence of uncertainty in action effects and sensing can lead to failure to execute plan actions. Therefore, more robust planners were developed to address the challenges of real-world planning domains. Sensor-based planners, such as GPT [17], MBP [10], and PTLplan [82], are designed to handle domains where information about the initial state might be incomplete, but more information can be acquired at execution time through sensing actions.

In the rest of this section, we give an overview of the two sensor-based planners PTLplan (developed by Karlsson [82]) and PC-Shop (developed by the author and Karlsson [20, 21]) that we used to compute solutions for dealing with unexpected situations resulting from lack of information when executing a task plan. Each of the planners is an extension of a classical planner. PTLplan is an extension of the forward-search planner Tlplan developed by Bacchus and Kabanza [3], while PC-Shop extends the hierarchical planner Shop developed by Nau and others [107].

Tlplan and Shop were demonstrated to be efficient planners because they made it possible for the user to provide control knowledge to prune search. PTLplan and PC-Shop were developed to take advantage of the efficiency of their predecessors in domains with the most general uncertainty conditions, i.e., stochastic outcomes of actions and partial observability. Probabilities are used to model the uncertainty about action outcomes and the world state. As a result, the planners are able to find plans that achieve the goal with a certain probability, whereas non-probabilistic planners are restricted to find plans that achieve the goal with certainty; such a plan might be hard to find or even non-existent.

3.1.1 Representation

PTLplan and PC-Shop utilize the same representation of actions and world states. Being a hierarchical task network (HTN) planner, PC-Shop also uses methods that describe how to refine abstract tasks into more detailed ones. The underlying representation for actions and states uses a rich LISP-style syntax language that supports among other structures: conditionals, probabilistic effects, quantified formulas, and partial state description.

Ground States

The state of the world is described in terms of fluents (state variables) and their values. A fluent literal has the form $(f\ t_1 ... t_n{=}v)$, denoting that the fluent f with parameters $t_1, ..., t_n$ has the value v (which might be non boolean). If the value is omitted, it is assumed to be `t` (true). The terms t_i can be constants, variables, or functional, where other non-boolean fluents may serve as functions. Examples of fluent literals are `(room r1)` and `(robot-in = r1)`. A fluent formula is a logical combination of fluent literals using the standard connectives and

quantifiers. Besides fluents that are stored directly in states, it is also possible to define axioms in terms of fluents (using fluent formulas and/or special forms for computing the value of the axiom). Fluents evaluated by external function calls as well as fluents derived from other fluents are also supported. We write $s \models \alpha$ to denote that the formula α holds in state s.

Observations

In partially observable environments, agents might not be able to determine the exact state of the world. Nevertheless, they are generally able to make observations that help them gain some information about their states. Observations in our model have the form of sets of ground fluent literals. We denote the set of all observations with O. An observation may contradict the actual state; thereby, uncertain or faulty sensors can be represented. For instance, the observation {`(open door1 = t)`} might occur when the fluent `(open door1 = f)` holds.

Belief States

Belief states are used to model uncertainty about the state of the world at a certain point in time. Formally, a belief state b is a triple $\langle S_b, P_b, O_b \rangle$ where S_b is a set of ground states s, P_b is a probability distribution over $s \in S_b$, and O_b is a set of observations the agent can make at runtime. The global probability $p(b)$ of being in a belief state b is defined as the sum of the probabilities of its element states, i.e., $p(b) = \sum_{s \in S_b} P_b(s)$. For instance, the belief state b representing the location of a robot as either room `r1` with probability 0.8, or room `r2` with probability 0.2, after observing a red light, is given as follows:

$$b = \langle S_b, P_b, O_b \rangle$$

where

$$\begin{array}{lcl} S_b & = & \{s_1 = \{\texttt{(robot-in = r1)}\}, s_2 = \{\texttt{(robot-in = r2)}\}\} \\ P_b & : & p_b(s_1) = 0.8; \quad p_b(s_2) = 0.2 \\ O_b & = & \{\texttt{(red-light)}\} \end{array}$$

Actions

An action specifies a transition from one state s to a non-empty set of new (alternative) states with associated probabilities. More formally an action a is triple $\langle prec_a, c_a, t_a \rangle$, where:

- $prec_a \subset S$ specifies in which states a is applicable.
- $c_a : S \longrightarrow \mathbb{R}$ is the cost function. We write $c_a(s_i)$ to refer to the cost of executing a in state s_i.

- $t_a(s_i, s_j, O')$ is the *state to state/observations* transition function such that s_i, s_j are states and O' is a set of ground observations. The function t_a encodes a probability distribution over the resulting states and their associated sets of observations when action a is executed in a particular state. In other words, $t_a(s_i, s_j, O')$ denotes the probability of ending up in state s_j and making the set of observations O' when action a is executed in state s_i.

In practice, actions are represented succinctly as parameterized action templates written in a language that includes constructs for conjunctive, conditional, stochastic and universally quantified effects. The language can also be used to specify ramification effects, which are associated not with a specific action, but are triggered when values of specific fluents change. We omit the details of how these templates are represented, as they are not significant for understanding the planning algorithms. It suffices to say that they can represent arbitrary cost and transition functions over a finite set of fluents.

Example The following action template encodes the movement of a robot from an initial room, identified by the variable `?r1`, to another room, identified by the variable `?r2`.

```
(ptl-action
   :name     (move ?r1 ?r2)
   :precond  (((?r1) (room ?r1)(robot-in = ?r1))
              ((?r2) (room ?r2)(connected ?r1 ?r2)))
   :results  (cond ((exists (?d)(door ?d)
                       (and (connects ?d ?r1 ?r2)(open ?d)))
                    (alt (0.8 (and (robot-in = ?r2)
                                   (obs (red-light))))
                         (0.2 (robot-in = ?r1))))
                   ((true)(robot-in = ?r1))))
```

This template specifies that in order for the movement action to be applicable, the robot has to be in room `?r1` and that `?r1` is connected to `?r2`. The results of the action are specified by a conditional construct that specifies that if there is an open door that connects the two rooms, the robot will move to `?r2` (with probability 0.8) and see a red light or stay unintentionally in `?r1` (with probability 0.2). Otherwise, the robot will stay in the starting room, i.e., `?r1`. Notice that the `alt` form encodes alternative effects, each with a probability of occurrence. The `obs` form is used to specify making observations.

An action $a = \langle prec_a, c_a, t_a \rangle$ is applicable in a belief state b if and only if it is applicable in each elementary state $s \in S_b$. The result is a set of belief states. $Result(a, b)$ denotes the set of the resulting belief states when action a is applied in the belief state b.

Belief state update

Let $b' \in Result(a, b)$, then $b' = \langle O_{b'}, S_{b'}, P_{b'} \rangle$ satisfies the following conditions:

- The belief state b' consists of the states reached by a transition with a specific set of observations $O_{b'}$:

$$s' \in S_{b'} \iff \exists s \in S_b : t_a(s, s', O_{b'}) > 0 \tag{3.1}$$

- The probability of each state $s' \in S_{b'}$ is determined by the probability of its predecessor states and the probability of their transitions to s':

$$P_{b'}(s') = \frac{\sum_{s \in S_b} P_b(s) \cdot t_a(s, s', O_{b'})}{\eta} \tag{3.2}$$

  The denominator η in equation (3.2) is a normalizing factor, and is defined as $p(b'|b, a)$, which is given next.

- The posterior probability for b' when action a is taken in b is:

$$p(b'|b, a) = \sum_{s' \in S_{b'}} \sum_{s \in S_b} P_b(s) \cdot t_a(s, s', O_{b'}) \tag{3.3}$$

- The expected cost of applying a in b is:

$$c_a(b) = \sum_{s \in S_b} P_b(s) \cdot c_a(s) \tag{3.4}$$

Planning Problem

A partially observable planning domain D is a triple $\langle S, O, A \rangle$ consisting of the state space S, the observation space O, and the set of actions A. A planning problem for a partially observable domain is a tuple $\langle D, b_0, g, succ_{min} \rangle$ consisting of a domain D, an initial belief state b_0, a goal formula g and a minimal success probability $succ_{min}$.

Plans

Both planners PTLplan and PC-Shop generate plans that have conditional form, i.e., having the structure of an "*if-then-else*" program. The conditions to branch on are expressed as conjuncts of ground observation-fluents. They represent possible observations that the agent can make at runtime. Therefore, conditional plans are a form of a closed-loop control strategy. The syntax of the generated plans is as follows:

plan ::= (*action* end-step*)
end-step ::= :success | :fail | (cond *branch**)
branch ::= (*obs-cond action* end-step*)
action ::= (*action-name term**)
obs-cond ::= *fluent-literal* | (and *fluent-literal**)

The term *action* represents an instantiated domain action, *obs-cond* is a conjunction of fluent-value formulae defined over observations, and :success, :fail are used to denote predicted plan success and failure respectively. This grammar accepts plans that have the structure of a tree where nodes with one successor represent actions, and nodes with several successors represent a conditional branching. The following plan is an example of a plan generated to enter a destination room, identified by the symbol r1, through door d1. The plan is generated starting from an initial belief state where the status of the door, i.e., open or closed, is not known. The plan includes an information gathering action to check the status of the door to decide what to do next:

```
((go-near d1)
    (check d1)
       (cond ((open d1 = t) (enter r1) :success)
             ((open d1 = f) :fail)))
```

The semantics of a conditional plan can be interpreted as applying the first action of the plan in the initial belief state. The second action is applied in the resulting belief state of the first action, and repeatedly applying an action in the resulting belief state of its preceding action. If the application of an action results in more than one belief state, then the subsequent action must be a conditional plan (cond $(c_1\ p_1) \ldots (c_m\ p_m)$) with as many branches as resulting belief states. Each branch $(c_i\ p_i)$ represents a contingency plan p_i to be executed in the belief state whose observations satisfy the branch condition c_i. Belief states at a certain execution time are uniquely identified by their observations. Therefore, the branch that has its observation fluent formula verified in the real world is selected for subsequent execution. At execution time, there must be at most one belief state whose observations are verified in the real world.

3.1.2 The PTLplan Planning System

PTLplan is a progressive planner that builds on a classical planner developed by Bacchus and Kabanza called TLplan [3]. PTLplan starts with an initial belief state, a set of actions, a goal formula, and a required minimum probability of success. Its output is a plan that when applied in the initial belief state, it leads to a belief state where the goal formula is satisfied with the minimum probability of success. The search process of PTLplan starts from the initial belief state and adds actions until a belief state satisfying the goal is reached.

When an action results in several new belief states with different sets of observations, the planner inserts conditional branches in the plan and continues planning for each branch separately. In order to search more efficiently, the planner can also eliminate belief states that invalidate a given temporal logic formula.

Strategic knowledge

In order to eliminate unpromising plan prefixes and reduce the search space, PTLplan utilizes strategic search control knowledge. This strategic knowledge is encoded as expressions (search control formulas) in first-order linear temporal logic (LTL) [45] and is used to determine when a plan prefix should not be explored further. One example could be the condition "never pick up an object and then immediately drop it again". If this condition is violated, that is it evaluates to *false* in some state, the plan prefix leading to that state is not explored further and all its potential extensions are pruned from the search space. A great advantage of this approach is that one can write search control formulas without any detailed knowledge about how the planner itself works; it is sufficient to have a good understanding about the planning domain.

LTL and extensions

LTL is based on a standard first-order language consisting of predicate symbols, constants and function symbols and the usual connectives and quantifiers. In addition, there are temporal modalities that are interpreted over a sequence of states, starting from the current state. For the purpose of PTLplan, they can be interpreted over a sequence of belief states $B = \langle b_1, b_2, \ldots \rangle$ and a current belief state b_i in that sequence. The modal formula `(until` ϕ_1 ϕ_2`)` means that ϕ_2 holds in the current or some future belief state, and that ϕ_1 has to hold in between; `(always` ϕ`)` means that ϕ holds in this and all subsequent belief states; `(eventually` ϕ`)` means that ϕ holds in this or some subsequent belief state; and `(next` ϕ`)` means that ϕ holds in the next belief state b_{i+1}. The following LTL formula specifies that the robot should not enter a room and then immediately return to its starting location:

```
(not (and (robot-in = r1)
          (next (and (robot-in = r2)
                     (next (robot-in = r1))))))
```

In addition to the temporal modal operators from LTL, PTLplan also uses a certainty operator `(nec` φ`)` denoting that φ holds in the current belief state b_i and a dual plausibility operator `(pos` φ`)` = $\neg$ `(nec` $\neg\varphi$`)`. This extension is necessary as formulas need to evaluated in belief states.

The following modal formula specifies that if it is possible that there is gas leakage in a room `r1`, then the robot must not switch off or on the light in that room:

```
(not (and (pos (gas-in r1))
          (switch r1 = s)(next (not (switch r1 = s)))))
```

There is also a goal operator `goal`, which is useful for referring to the goal in search control formulas. `(goal` φ`)` denotes that it is among the agent's goals to achieve the fluent formula φ. Semantically, this modality will be interpreted relative to a partial state g representing the set of states that satisfy the goal. Finally, `(obs` φ`)` means that φ was observed in the current observation o_i. Fluent formulas are restricted to appear only inside the `nec, pos, goal` and (for atomic fluent formulas) `obs` operators.

In order to efficiently evaluate control formulas, PTLPLAN incorporates a progression algorithm (similar to the one of TLPLAN [3]) that takes as input a formula f and a belief state b and a goal state s_g (a ground state) and returns a formula f^+ that is "one step ahead", i.e., corresponds to what remains to evaluate of f in subsequent belief states. Thus, the progression algorithm can be applied, during planning, for each transition to a new belief state to have an up-to-date formula representing what remains to evaluate. If that turns out to be *false*, the formula has been violated and search does not continue from that belief state.

3.1.3 The PC-SHOP Planning System

PC-SHOP, which stands for "Probabilistic Conditional SHOP", follows the success of the classical HTN planners SHOP [107, 108] and SHOP2 [109] (developed for deterministic and fully observable domains). HTN planning is one of the oldest and most well-tested approaches to planning [141, 151, 38]. It allows one to code domain-dependent knowledge in a powerful way through procedures that describe how to solve the planning problem, resulting in more efficient search and better support for large domains. SHOP is known to be a simple but high-performing HTN planner, where tasks are planned for in the same order that they will be executed.

The objective of developing PC-SHOP is to take advantage of the efficient hierarchical planning techniques to deal with the inherent complexity of planning in uncertain domains. Besides hierarchical planning under uncertainty being interesting in its own right, PC-SHOP was developed for solving actual robotic problems. One example is recovery from ambiguous cases in perceptual anchoring, where it was applied to solve some of the problems reported in chapter 7.

An HTN planner works to solve abstract tasks: each goal and subgoal is represented as a task. Tasks can be either primitive or composite. Primitive tasks correspond to executable actions, while composite tasks can be decomposed to

a task network consisting of a set of more detailed tasks with some ordering constraints. The different ways a task can be decomposed into task networks are specified by methods. For instance, building a house can be considered as a composite task. A method for building a house might be: set the foundation, build the outer walls, build the roof, and so on. Each of these sub-tasks might in turn be subdivided into further subtasks. There might also be alternative methods for solving a certain task.

Methods

In PC-SHOP, methods are used to control search and they provide the knowledge of how to decompose abstract tasks into more detailed lists of tasks. A method has the following form:

$$(h \quad p_1\ t_1\ p_2\ t_2 \ldots p_n\ t_n)$$

where, h is the method's name and should unify with an abstract task. p_i is a list of precondition formulae. Each precondition formula is a list of clauses of the form $(\overline{v}, \theta, \varphi)$, where $\overline{v}$ is a list of variables that need to be bound, θ is an atomic fluent formula containing $\overline{v}$, and φ is a fluent formula or a modal formula (`nec` or `pos`). The keyword `:first` can be inserted in the beginning of a precondition, to signify that only the first found variable binding should be used, i.e., the first binding of the variables of that precondition that satisfies the entire precondition list.

The task lists t_i are recursively built by tasks and the constructs `(:unordered` $t_1\ t_2 \ldots$`)`, `(:ordered` $t_1\ t_2 \ldots$`)`, and `(:cond (`$o_1\ t_1$`) (`$o_2\ t_2$`) ...)`. The key words `:ordered` and `:unordered` specify whether tasks can be interleaved or not. The label `:immediately` can precede a task to mean that when the preceding tasks have been processed, the task so labeled must be processed at once. The key word `:cond` is used to handle feedback and conditional branching. It specifies that a branch should be generated for each pair $(o_i\ t_i)$ if the last action resulted in a belief state whose observations satisfy o_i; the branch includes the tasks in t_i.

Semantically, a method specifies that the abstract task h is further decomposed to the tasks in t_k if the precondition p_k holds and all $p_{j<k}$ are false in the axiom set and all the ground element states composing the current belief state. It is worth noting that several methods might have the same name to specify different ways of decomposing the abstract task h.

Example The following method specifies how to decompose a task of going to a location `?des` into more detailed tasks:

```
(method (!goto ?des)
   (((?src)(robot-in = ?src)(same-floor ?src ?des)) )
```

```
(:ordered (move2door ?des)(enter ?des))
(((?fl)(floor ?des = ?fl)))
(:ordered (call-lift)
          (:cond ((lift = OK) (use-lift ?fl))
                 ((lift = NOK)(use-stairs ?fl)))
          (!goto ?des)))
```

Briefly, the idea behind the method is that whenever the robot is on the same floor as the destination `?des`, the decomposition includes the ordered primitive tasks of moving to the entrance of `?des` and then entering `?des`. Otherwise, the robot has either to take the lift or the stairs depending on whether the lift is working or not. The second decomposition includes a recursive call to the same method once the robot reaches the floor of the destination.

The Planning Algorithm

PC-SHOP is a total order forward-search algorithm. It gets as input a set of belief states BS (initially one), an ordered list of tasks to achieve T, a minimum probability of success, and a domain description D. The output is a plan, composed of only instantiated primitive tasks, with a success probability greater or equal to the minimum one. The algorithm recursively decomposes the tasks in T, taking into account ordering constraints, until it finds a plan that contains only primitive tasks. The process of task decomposition might result in different sub-tasks because more than one method might be applicable when a composite task is decomposed. Therefore, the algorithm backtracks whenever a decomposition does not contribute to finding a plan that solves the given planning problem.

3.2 Description Logics

In this section, we give a short overview of description logics, which are classical AI formalisms used for knowledge representation and reasoning purposes. Our particular focus will be on presenting the LOOM system [97].

Description logics (DLs) (see the book by Baader *et al.* [2]) are decidable fragments of first order logic intended for knowledge representation and management. They are used to represent domain knowledge of applications through the specification of the domain concepts and relationships between concepts (also called terminology). The description of the world consists of assertions of properties and relations between individuals present in the domain[1] .

An important characteristic of description logics is their reasoning capabilities of inferring implicit knowledge from the explicitly represented knowledge.

[1]In DLs terminology, the term TBOX, respectively ABOX, is used to refer to terminological knowledge, respectively assertional knowledge, specified in the domain knowledge-base.

In DL formalisms, unary predicates represent concepts (also called classes), i.e., sets of individuals (also called objects), and binary predicates express relationships between individuals. Concept expressions can be built using a small set of connectives and constraints over the individuals that are in a relationship with a specific individual. Concepts that are not defined in terms of other concepts are called atomic concepts.

In this thesis, we employ description logics to encode and reason about semantic domain-knowledge for the purpose of plan execution monitoring (see chapter 4). The major advantages of using description logics (DLs) are as follows:

- DLs provide a concise representation of the world, as they can express general knowledge about classes of objects. Thus a lot of information can be kept implicit. For instance, one does not have to state explicitly that room `r5`, which is an office, contains a desk. Such information can be inferred from the general description of the class 'office'.
- DLs are fairly expressive yet supported by efficient inference mechanisms, making them practically useful.

The LOOM System

In practice, we use LOOM [97], a well established knowledge representation and reasoning system for modeling and managing semantic domain-knowledge. The choice of LOOM was suggested by practical considerations: mainly because it is a well supported open source project[2]. LOOM provides a definition language to write definitions of concepts and relations as well as constraints over them. An assertion language is also provided to assert facts and constraints about individual objects.

Knowledge in LOOM is organized in knowledge bases that contain two types of knowledge: terminological and assertional. The terminological knowledge is referred to as the TBOX. It contains definitions of concepts and relations between concepts. The assertional knowledge is referred to as the ABOX and it contains assertions about individual objects. In other words, the ABOX represents an instance of the TBOX describing a possible state of the world.

Concepts are used to specify the existence of classes of objects, such as "there is a class of rooms" or "a bedroom is a room with at least one bed":

```
(defconcept room)

(defconcept bedroom
            :is (:and room (:at-least 1 has-bed)))
```

[2]The work described in this thesis can be implemented using other knowledge representation and reasoning systems based on description logics.

Atomic Construct	FOL
`A`	$A(x)$
`(some R C)`	$\exists y.\,(R(x,y) \wedge C(y))$
`(all R.C)`	$\forall y.\,(R(x,y) \rightarrow C(y))$
`(oneOf` $a_1, \ldots, a_n$`)`	$(x = a_1) \vee \ldots \vee (x = a_n)$
`(at-least n R)`	$\exists y_1, \ldots, y_n.\, \bigwedge_{1 \leq i \leq n} (R(x,y_i)) \wedge \bigwedge_{1 \leq i < j \leq n} y_i \neq y_j$
`(at-most n R)`	$\exists y_1, \ldots, y_{n+1}.\, \left(\bigwedge_{1 \leq i \leq n+1} (R(x,y_i))\right) \rightarrow \left(\bigvee_{1 \leq i < j \leq n+1} y_i = y_j\right)$
`(exactly n R)`	$\mathcal{FOL}(\geq n\ R) \wedge \mathcal{FOL}(\leq n\ R)$

Table 3.1: Some of LOOM's atomic concept constructs and their equivalent first-order logic formulas.

The first definition declares that there is an atomic class of items called `room`. The second definition introduces a class named `bedroom` whose instances (elements) are objects of type `room` in which there is at least one `bed`. A concept can also be specified as *primitive* to reflect that its definition is not completely specified, i.e., there are constraints that are not represented for individuals of that concept.

The term `has-bed` in the second definition specifies a relation between objects of class `bedroom` and objects of class `bed`. This relation is defined in LOOM as follows:

```
(defrelation has-bed
             :domain bedroom
             :range  bed)
```

The construct `(:at-least 1 has-bed)` specifies a constraint over the number of beds that can be in a bedroom. Number constraints are used to specify the number of objects of one class that are related to another object of another or the same class. It is also possible to specify constraints over the types of objects an object can be in relation with. Note that `(:at-least 1 has-bed)` itself defines a class denoting all objects that have at least one bed.

More complex concept expressions are constructed by combining other concept names using a limited number of connectives (`and, or, not, implies`). The semantics of concept expressions are interpreted in terms of set theory operations (intersection, union,...) or in terms of equivalent first-order logic formulas over a non empty set of individuals. Table 3.1 shows some of LOOM's atomic concept constructs and their equivalent first-order logic formulas.

Once the general semantic knowledge is constructed, specific instances of classes can be asserted to exist in the real world. For example:

```
tell (bedroom r1)(has-bed r1 b1))
```

asserts that `r1` is an instance of `bedroom` and results in classifying `b1` as an instance of the class `bed` because the range of the relation `has-bed` is of type `bed`. The instance `r1` is also classified (deduced) automatically as an instance of the class `room`.

Classification is performed based on the definitions of concepts and relations to create a domain-specific taxonomy (figure 3.1 shows a taxonomy for classifying house furniture items). The taxonomy is structured according to the superclass/subclass relationships that exist between entities. When new instances of objects are asserted (added to the knowledge base), they are classified into that taxonomy.

LOOM Supports a first-order query language to retrieve instances from a knowledge base. It is also possible to ask the knowledge base whether or not a proposition is true. LOOM uses open-world semantics as the default assumption when trying to prove or disprove a proposition. This makes it possible to conclude whether the truth value of a proposition is true, false, or simply unknown. For instance, one might ask LOOM whether the instance named `r1` is a room by issuing the following question:

```
(ask (room r1))
```

To ask if the instance named `r1` can be proved to be *not* a room, one can issue the following question:

```
(ask (not (room r1)))
```

3.3 Robot Architecture

The aim of this section is to describe the integrated mobile robotic architecture that we used in our test scenarios. The architecture comprises two main layers: the bottom layer is a behavior-based navigation layer, while the top layer constitutes a deliberation layer. Figure 3.2 gives an overview of the overall architecture and how both layers are integrated. It is worth mentioning that by no means does the work presented in this thesis rely on a specific robot architecture. The only requirement of our work is that the underlying architecture includes an executor of symbolic plans.

The behavior-based layer is implemented by the ThinkingCap (TC) mobile-robot control architecture developed by Saffiotti [131, 133]. Its main task is the control of the robot platform to perform navigation tasks. It is a hybrid control architecture comprising a fuzzy-logic behavior based controller and a navigation planner.

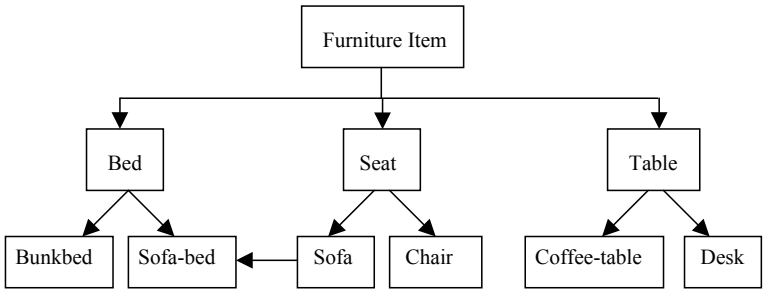


Figure 3.1: Part of a taxonomy of some classes of furniture items.

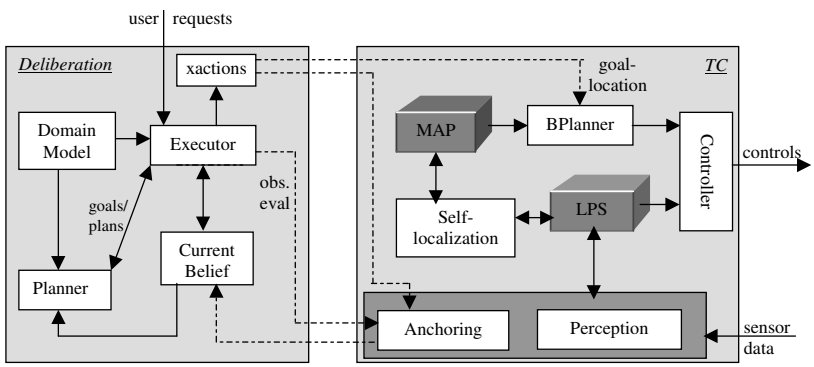


Figure 3.2: Global robot architecture used in our test scenarios. The architecture is composed of a deliberation layer that sits on top of a behavior-based navigation layer.

The deliberation layer includes all the necessary functionalities needed to generate and execute high-level symbolic task-plans. The plan executor is designed to handle probabilistic conditional plans and is not constrained by using a specific planner. In fact, the executor can execute any plans fulfilling some representational constraints, mainly the syntax of the plans, and the specification of how to execute the plan actions. The execution system uses three hierarchical layers each with a specialized process. The top layer manages high level plans, user requests, and recovery when necessary. The middle layer has a more specialized process whose task is to execute the actions of the plan selected by the upper layer, whereas the third layer is in charge of low-level execution and monitoring. The overall system has been successfully used for research on sensor-based planning for mobile robots, most notably in the areas of perceptual anchoring as reported by Broxvall *et al.* [26] and active smelling in the work by Loutfi *et al.* [91].

3.3.1 Behavior-based Architecture

The ThinkingCap (TC) robot control architecture [131, 133] controls the mobile robot using fuzzy behaviors expressed as sets of control rules. It integrates navigational capabilities that help in creating, executing and monitoring navigation plans. The main components of TC are described briefly in the following paragraphs.

B-Planner The Behavior Planner is a backward search planner that is designed to generate navigation plans. It computes a set of context-behavior rules having the form `IF context THEN behavior`, where `context` is a formula of fuzzy predicates evaluated on the current world model (LPS below). The context-behavior rules of a B-Plan are evaluated in parallel, influencing the overall robot behavior according to the combined value of their respective context.

LPS The Local Perceptual Space (LPS) is used to store information about the world around the robot expressed as object descriptors and perceptual data given in robot's coordinates. It contains raw sensor data as well as perceptual features such as lines and openings.

Controller This component is in charge of generating crisp control values (steering and velocity) through defuzzification of the result of the combined active fuzzy behaviors (according to their context calculated from the LPS).

Map The Map encompasses the global map of the environment. It is basically a topological map with nodes describing metric sectors and edges representing connection gateways.

Self-localization The self-localization component is used to compute the robot's position with respect to the map.

Anchoring and perception The anchoring module provides an interface to perceptual information from the sensor systems of the mobile robot. It contains a number of functionalities for establishing the connection between the high-level symbolic representation and low-level perceptual representations such as video camera images. The anchoring module can also provide information about already perceived objects, such as position and visual features extracted from the LPS. More about the anchoring process is given in chapter 7.

Example of B-Plans. Consider the map of the environment depicted in figure 3.3 where the robot is located in the corridor `C1`. To enter room `r1`, the B-Planner is asked to generate a navigation plan that achieves the fuzzy goal `(at me r1)`. The following B-Plan is generated to achieve the goal where each row represents a fuzzy rule:

(in me r1)	still(goal)
(at me r1) (not (in me r1))	reach(r1)
(at me c1) (open door d1) (facing me d1) (not (at me r1))	cross(d1)
(near me d1) (oriented me d1) (not (at me r1)) (at me c1) (open door d1) (not (facing me d1))	face(d1)
(at me c1) (traversable c1 d1) (oriented me d1) (not (near me d1))	follow(c1)
(not (near me d1)) (at me c1) (traversable c1 d1) (not (oriented me d1))	orient(d1)

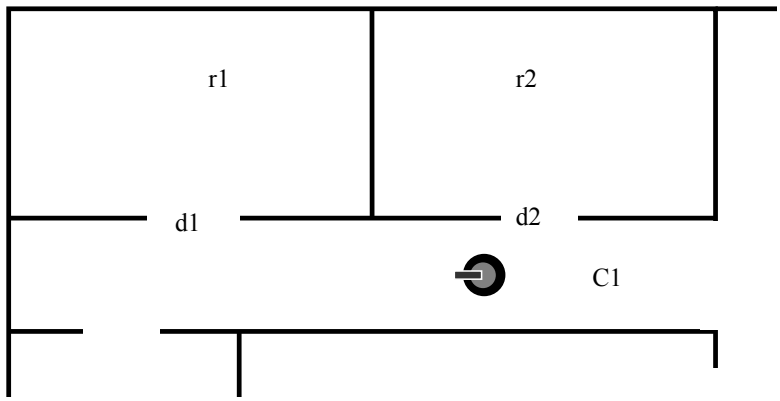


Figure 3.3: An example of an environment where a robot is trying to navigate from its current location in the corridor `C1` to room `r1`.

3.3.2 Execution and Monitoring of Conditional Plans

In this section we outline the main processes involved in executing a high-level conditional plan and its actions. The executor uses different data structures to manage the execution of multiple plans that can arrive asynchronously.

Each plan has an execution context that includes the initial belief state, the goal, the plan itself, the last action executed, and the priority of the plan. The execution context is placed in one of three queues waiting for execution. The queues are associated with classes of plans identified by their priorities. A plan can have either low, medium, or high priority.

Plan execution is a hierarchical process that follows the semantics of conditional plans described in section 3.1.1. Figure 3.4 depicts the three sub-processes forming the hierarchy. At the top-level, there is a state-machine process in charge of selecting the plan with the highest priority for execution. It is also in charge of launching the recovery of plans when one of their actions fails to execute. At the second level, a more specialized state-machine process is used to control the execution of the actions of conditional plans, reporting the outcome of the action to the high-level process. The action execution process is mainly used to extract the execution procedures of the current action to be executed. The procedures specify the different steps needed to achieve the effect of the corresponding plan action. The action execution process launches the appropriate processes to execute and monitor the progress of the steps.

Plan Execution Process

The plan-execution process is launched upon starting up the robot. While in state INIT, the process checks periodically for waiting plans, proceeding with the execution of the plan with the highest priority. The actual execution of a plan

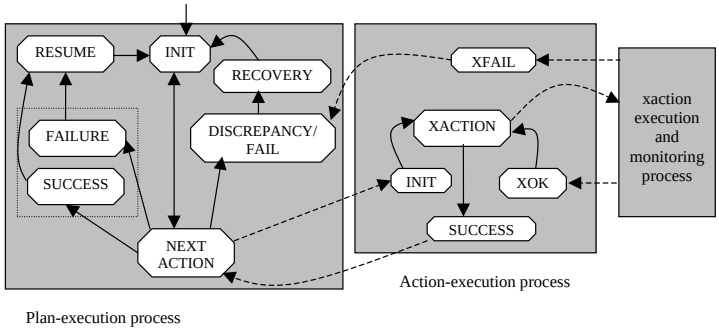


Figure 3.4: The different processes in charge of executing conditional plans of partially-observable domains.

starts in state NEXT-ACTION, where the executor checks the type of the current action selected for execution. In this context, an action can be either `:success`, `:fail`, a conditional plan, or a simple domain action. It is also in this state that plans, with higher priority, can interrupt the execution of the current plan. As mentioned earlier in section 3.1.1, the special action `:success`, respectively `:fail`, reflects the planner's prediction that the generated plan will succeed, respectively fail. If the plan reaches a predicted failure, then the process simply drops the plan. Reaching a predicted success state means that the plan has achieved its goals with success. Because the execution of the currently dropped or successful plan might have interrupted the execution of another plan, with lower priority, the execution process checks whether there is an interrupted plan waiting for execution in order to restore its execution context and start it again (state RESUME).

One issue that might arise is the inability to resume the execution of the interrupted plan because its next action to execute is not applicable in the current real-world situation as a result of execution an interrupting plan. To partly remedy this problem, the process does not interrupt an executing plan unless it can find a chaining plan that ensures that the interrupted plan can be resumed when the interrupting plan finishes execution with success. It is worth noting that finding a chaining plan might be problematic since the interrupting plan can have more than one branch that leads to success. Generating the chaining plan would take into account this issue; thus, chaining plans are generated for branches whose final belief states satisfy the goals of the interrupted plan. Upon resuming the execution of an interrupted plan, the process executes the chaining plan first, and then the rest of the interrupted plan. Obviously, this works only when the interrupting plan has been successfully executed. In case of failure, the planner is asked to find a chaining plan that when starting from the current state, it achieves the preconditions of the first action of the rest of the interrupted plan.

In the NEXT-ACTION state, in figure 3.4, the current action might also be a conditional plan. In that case, the process checks the contingency condition for every branch in the real-world state, and chooses the branch whose condition is verified in the real-world observations. Evaluating the branching conditions is performed by calls to specialized procedures to evaluate the condition fluents using the perceptual data provided by the anchoring module. If, on the other hand, the current action is an instantiated domain action, the process checks its preconditions in the current belief state. In case the preconditions are satisfied, another process is launched to execute the action as described in the next subsection. In case of a discrepancy, the process calls special functions that estimate the current belief state so that more information is included about the current situation. Then, the planner is called to find a plan that achieves the preconditions of the failing action (state RECOVERY).

Evaluation of Observation Fluents

To be able to evaluate observation fluents at execution-time, a procedure must be defined and associated with each observation fluent. When executing a plan, the executor uses the observation fluent to determine the evaluation procedure associated with it. The procedure specifies how to evaluate the observation fluent by calling TC's fuzzy observation predicates such as `(open d)` (which refers to the degree to which the door `d` is open). The procedure might also use data stored in the LPS to evaluate observation fluents not computed by TC. For instance, the metric data in the LPS can be used to evaluate whether two perceived objects `obj1` and `obj2` are near each other, i.e., to establish the truth value of the fluent `(near obj1 obj2)`.

Executable Actions

For each plan action, the user creating the planning domain provides the different executable actions `xactions`, which are defined in terms of the functionalities of the robot control-architecture (in this case TC). Typically, an executable action defines a procedure that calls TC functions to produce behaviors that would achieve a specific low-level goal. The procedure also defines the monitoring process to be associated with the execution of the behaviors in order to make sure to respond to unexpected events and apply local recovery strategies if possible. At this level, the monitoring process is a hard-coded procedure that is tailored to the `xaction` it is in charge of. Its main task is to give an indication of whether the execution of the `xaction` has been successful or it failed, so that a deliberate recovery would be considered for the high-level action.

Example In order to execute the high-level action `(enter r1)` to enter room `r1`, the execution part consists of a procedure "`execute-enter (room)`" that (1) calls the B-Planner with the goal `(robot-in r1)` where the goal

represents a fuzzy predicate, and (2) installs a monitor process for the generated B-Plan. In case of a failure to achieve the B-plan goal, the monitor calls the B-Planner to replan for another navigation path to enter room `r1`.

Action Execution Process

The execution of high-level actions is performed by a more specialized process whose states are outlined in figure 3.4. Activating action execution at this level involves blocking the launching process, i.e., the plan-execution process. High-level action execution starts by retrieving the `xactions` one at a time (state `xaction`). As we outlined before, there are specialized procedures for each `xaction` specifying the necessary steps to perform along with a monitoring process. Calling the specialized procedure of an `xaction` results in blocking the action-execution process and launching the monitoring process of the `xaction`.

The monitoring process of an `xaction` can respond to failures by calling precomputed procedures or by calling the B-Planner to find another local B-plan. The process has also to guarantee that the blocked action-execution process is notified about the outcome of the execution of the `xaction`. If the execution of the `xaction` is successful, then the action-execution process is awaken in the state XOK, otherwise it is awaken in the state XFAIL.

Awakening the action-execution process in state XFAIL is an indication of the inability of the robot to execute the `xaction` with success, and therefore leads to the failure of the high-level action. Thus, the plan-execution process is notified in turn that the execution of the current action has failed (state DISCREPANCY/FAIL). If, on the other hand, the monitor of the `xaction` reports to have executed `xaction` successfully, the action-execution process repeats the same steps with the remaining `xactions`. When all the `xactions` have been successfully executed (state SUCCESS), the action-execution process awakes the plan-execution process in the state NEXT-ACTION, so that the same steps can be performed with the next high-level action of the plan.

3.4 Robot Platform

The mobile robotic platforms that we used in our experiments were two Magellan Pro robots. The robots are named Pippi and Emil (see figure 3.5). Their bases measure $0.25m$ high and $0.4m$ in diameter. They are designed to be used as indoor research robots. They are driven by 2 motors attached to the two wheels of the robot. Both robots come equipped with the following sensors:

1. 16 sonars that can be used to detect nearby obstacles, within a range of $0.15 - 7.0m$.

2. 16 infrared sensors that have a shorter range: up to $0.5m$.

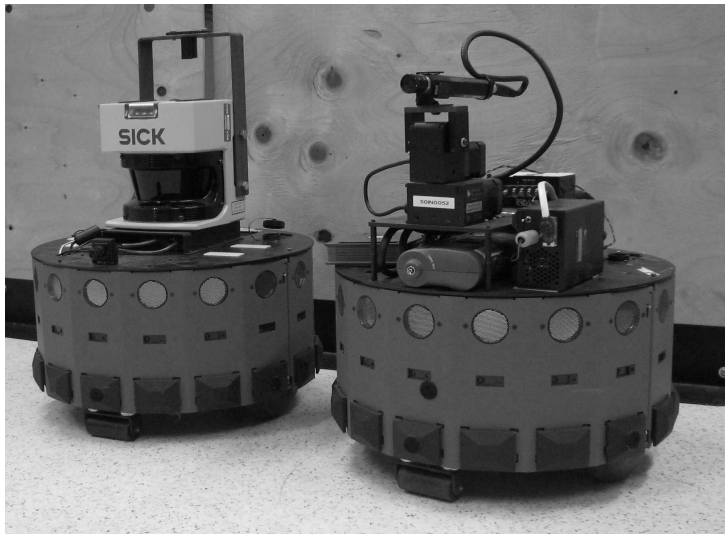


Figure 3.5: From left to right: Emil and Pippi, the two Magellan Pro mobile robots used in our test scenarios.

3. 16 tactile sensors that can be in a binary state, i.e., bumping or not bumping into an object.

4. Odometry sensors, on both driving wheels, used for measuring the planar distance traveled by the robot.

Other sensors were added to the robots in order to make other types of measurements. These include a color camera on both robots (the one on Pippi is attached to a pan-tilt unit), an electronic nose on Pippi, and a laser range finder on Emil.

3.5 Summary

This chapter presented the main tools that were used to develop and test our solutions to the problems addressed in this thesis. We would like to point out that among these tools, the sensor-based planner PC-SHOP and the hierarchical plan executor represent scientific contributions of the thesis. Starting from the next chapter, we will be focusing on our approach to robust execution of symbolic plans by indoor mobile robots.

Chapter 4
Monitoring of Implicit Expectations

Having reviewed research work related to monitoring the execution of plans and strategies for dealing with unexpected situations, it is now time to start presenting our own solutions to the problem addressed in this thesis. In this chapter, we present a novel approach for intelligently monitoring the execution of symbolic plans by mobile robots acting in indoor environments, such as offices and houses. The novelty of the approach lies in using domain knowledge to derive implicit expectations of executing actions successfully. The robot uses the immediately available perceptual information to check whether those expectations are met or violated.

As it has been discussed in chapter 2, approaches used to monitor plan execution have generally focused on using the explicit effects of actions as expectations that should be verified when the corresponding action is executed successfully. For instance, an explicit effect of grasping an object can be that the robot is holding that object, while the explicit effect of entering a room can be that the robot is inside that room. Explicit effects are generally extracted from the models of actions, which are given as part of the planning domains. Examples of such approaches include the ROGUE [69] mobile robotic architecture, the work by Fichtner *et al.* [50], and our hierarchical plan execution and monitoring system, which was described in chapter 2.

Relying only on explicit effects to monitor action execution supposedly means that the derived expectations are directly observable. For example, a mobile robot that has executed the planned action `(enter r1)`, to enter the living-room, would query its self-localization system to verify that the explicit expectation `(robot-in = r1)` holds. This way, execution monitoring completely relies on the accuracy of the self-localization system. Moreover, checking expectations in real-world environments is inherently a complex process that goes beyond checking what the robot directly senses.

This chapter proposes to increase the reliability of monitoring of plan execution by incorporating more advanced forms of reasoning. In particular, we propose to use semantic knowledge about the domain to derive *implicit expectations* about the effects of plan actions, and to monitor these expectations using the available perceptual information. By implicit expectations we mean expectations that can be logically derived from the explicit ones (the ones encoded in the action model) through the use of semantic knowledge. In the above example, if the action `(enter r1)` was successful, and since `r1` is an instance of the class `Living-Room`, the robot should expect to see objects that are typical of a living-room such as a TV-set and a sofa. If the robot sees an oven, it should conclude that it is not in the living-room, and henceforth that the execution of `(enter r1)` was not successful. As another example, if the executed action is to grasp a coffee cup, then semantic knowledge could be used to generate and check the implicit expectation that the object in the gripper has properties such as being a container and having exactly one handle. Therefore, checking implicit expectations when acting in indoor environments helps, among other things, to verify that the robot is in the correct room, and not (1) dislocated or (2) have an erroneous map. We also consider the implicit expectations to be details that would add complexity to the planning task if the task-planner has to reason about them. That is why they are encoded in a separate semantic knowledge base, i.e., outside the action models used by the task planner.

In this chapter, we present a monitoring approach that is intended for crisp domains, i.e., plans are composed only of actions with deterministic effects and where perceptual information is assumed to be reliable. In chapter 5, we describe another approach to handle the execution of nondeterministic actions and noisy sensing. Chapter 8 presents real-robot test scenarios where the approach was employed in monitoring the execution of indoor navigation plans.

The rest of the chapter is organized as follows. In the next section, we go through an illustrative scenario of a mobile robot acting in a house environment. Then, we describe how we encode and reason about semantic domain-knowledge using the LOOM system [97]. The details of the semantic knowledge-based execution-monitoring approach are presented in the section that comes next. Finally we conclude and discuss the contents of the chapter.

4.1 A Motivating Scenario

To better illustrate our ideas, we describe a scenario of a mobile robot that is acting in a house environment to accomplish a multitude of household tasks such as, cleaning the floor, serving drinks to guests, doing laundry, etc. Figure 4.1 shows a map of such a house where our robot can live and serve. The house comprises rooms of different types such as bedrooms, kitchen, etc., as well as objects of different types that can exist in the different rooms. Types of objects include sofas, beds, tables, chairs, kitchen appliance (silverware, utensils,...), plants, etc. The environment is not supposed to be specifically structured in a

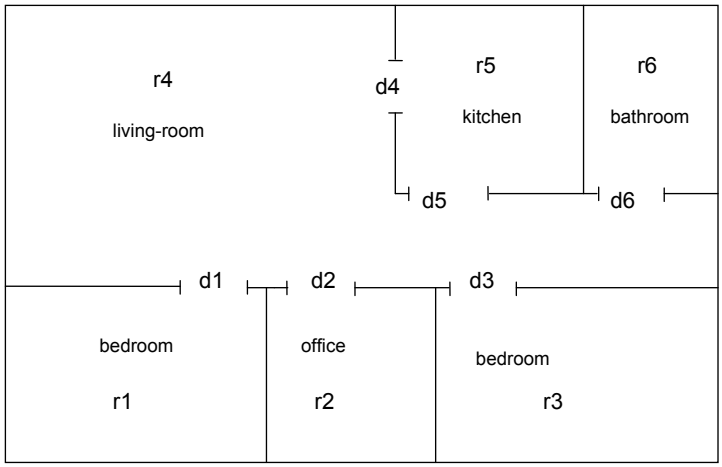


Figure 4.1: A plan of a house where a mobile robot can act to achieve household tasks.

way that makes it easy for the robot to act. The environment is also dynamic, as objects like chairs and cups can be displaced from one location to another either by the robot itself or by the humans living in the house without notifying the robot.

The robot is supposed to be acting autonomously and is equipped with functionalities that help it accomplish its tasks. These include low-level navigation and manipulation functionalities as well as high-level deliberation and problem solving capabilities. In particular, an on-board planning engine is used to synthesize plans that specify the necessary actions that need to be executed in order to accomplish a specific task. The robot is also supposed to have access to a knowledge base (KB) where information about its environment is stored.

Suppose that while the robot is busy cleaning the living room, it is asked to bring a cup of coffee immediately. To do so, the robot first suspends the task of cleaning and then calls its on-board planner to generate a plan that helps it accomplish the assigned task. The task planning system uses information about the current location of the robot as well as the domain knowledge (e.g., that cups are generally arranged in the cupboard, which is located in the kitchen) to generate a task plan that could include the following actions:

```
(goto d4)(enter r5)(open-cupboard cb1)
        (pick-up c1)(fill c1 coffee)
                    (goto d4)(enter r4)(deliver c1 person1)
```

where `r5` and `d4` are symbols denoting the kitchen and a door that leads to it, respectively, while the cupboard is referred to by the symbol `cb1`, and the cup

by the symbol `c1`. The final action specifies that the robot should deliver the cup of coffee to the person "`person1`" who ordered the cup of coffee.

If the robot started from a position where it was initially disoriented, then the execution of `(goto d4)` could result in the robot being not in front of door `d4` but in front of the door that leads to `r1` instead. A standard plan execution monitor checks that action `(enter r5)` has been executed successfully, i.e., the robot is in room `r5`, simply by checking the current location provided by the robot's self-localization system. Due to the initial error in orientation, the location of the robot can be erroneously computed to be `r5`. Using semantic domain-knowledge in execution monitoring makes it possible to check the implicit effects implied by being in `r5`. In particular, since `r5` is asserted to be a kitchen, the robot looks for indications that it is in a kitchen, such as seeing an oven, sink, or a stove. If, on the other hand, the robot sees a bed, it should conclude that it is not in the kitchen, but in a bedroom. Such information is derived from the semantics of the different rooms and objects present in the house. Similarly, the plan executor can check whether the execution of the action `(pick-up c1)` has succeeded not only by verifying that its gripper is holding "something", but also by checking that what is held satisfies the description of an object of type "cup".

4.2 Semantic Knowledge

As mentioned in chapter 1, semantic knowledge refers to the meaning of objects expressed in terms of their properties and relations to other objects. Objects that share the same properties and relations are grouped into classes (or concepts). For instance, objects of type room and with at least one bed are instances of the class bedroom, while rooms with sofas and TV sets define a class of living-rooms, etc. Such knowledge captures the way humans organize knowledge about objects as instances of general categories. Therefore, semantic knowledge can be used to help mobile robots communicate with humans. For instance, in the work of Theobalt and colleagues [142], a robot can ask humans about its location in terms of high-level descriptions of locations instead of using metric data. Semantic knowledge has also been used in other areas of mobile robotics, such as scene analysis by Hois and colleagues [73] and map building, e.g., Galindo *et al.* [60], Nuchter *et al.* [116], and Ekvall *et al.* [44].

Obviously, the semantic knowledge base should capture knowledge about objects that are part of the environment of the robot. At a first glance, the specification of such knowledge might appear to be an easy task. However, indoor environments are usually cluttered with objects of different types, which makes it difficult to provide knowledge about all of them. As our aim is to use semantic knowledge for the purpose of monitoring the execution of symbolic plans, we should be careful not to include knowledge about any type of objects but the ones relevant to the task at hand, i.e., execution monitoring. Therefore, our design of the semantic knowledge base takes into account the formal definitions

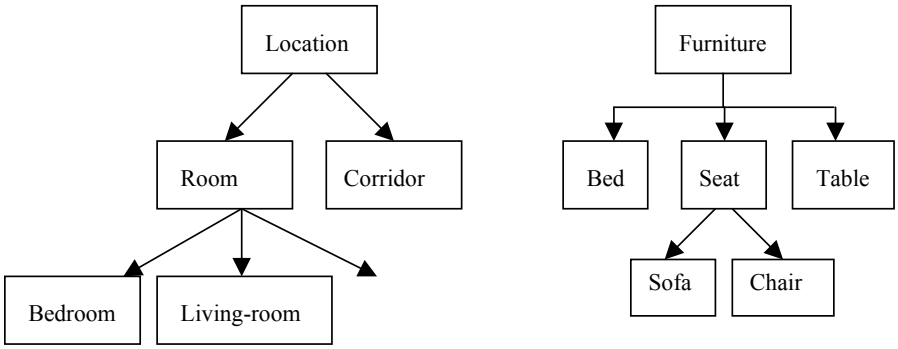


Figure 4.2: Parts of taxonomies of some classes of objects in a house environment. (Left) different types of locations. (Right) some types of furniture. The relationships between the classes of the taxonomies are not shown.

of the actions that form the planning domains. As a first step, we use the action models to identify the objects manipulated by the action. Then, we provide knowledge about the different types of those objects in terms of properties and relations to other objects. Next, we apply the same process to add knowledge about the related objects, and so on. For instance, in a navigation planning domain, the `(enter loc1)` action can be used to model the movement of the robot to enter a location `loc1`. Consequently, the semantic knowledge base includes knowledge about the different types of locations that exist in the robot's environment. These can be corridors, halls, and rooms. The knowledge base includes also knowledge about the different types of rooms (e.g., kitchen, office, etc.), and also the objects that are typical of such rooms, such as ovens, desks, etc. Figure 4.2 shows two taxonomies of classes of objects that can be provided starting from a navigation action that takes place in a house environment.

We restrict knowledge about objects to be in terms of properties that the robot can directly observe or are defined in terms of other observable properties. For example, the definition of an object of type cup might include that the object must be a container that has one handle, where container and handle are atomic concepts, hence they must be directly observable. Direct observability implies that it is the task of the perception module to tell whether a perceived object is an instance of an atomic class, e.g., whether a perceived object is a container.

4.3 Overview of the Approach

In this section, we give an overview of how semantic domain-knowledge can be integrated in the process of monitoring the execution of symbolic plans. The process is meant to be used for crisp domains where both sensing and

actions are assumed to be reliable. We start by describing the overall monitoring process of checking the effects produced by the execution of actions. Then, we give an overview of the components involved in deriving and checking the implicit expectations of executing plan actions successfully.

4.3.1 The Overall Monitoring Process

As explained in chapter 2, monitoring the execution of symbolic plans is a model-based process that compares the explicitly modeled effects of an action to the actual outcome produced by the execution of that action. The actual outcome of the action is computed using the perceptual information acquired by the robot's on-board sensors. One can also use the model of the executed action together with knowledge about the objects manipulated by the action to compute a set of expectations that are not explicitly encoded in that model. These expectations can then be verified using the same perceptual information used to check the explicit effects.

There are two possible ways of integrating the results of monitoring the two types of expectations. The first way is to monitor them separately and then combine the results to produce a final result about whether the execution of the action has succeeded. The second way is to use the result produced by the process of monitoring the implicit expectations as an additional check of the results produced by the process of monitoring the explicit effects of the action.

Using the first way implies having a mechanism that handles situations where the results of the two processes might be contradictory. For example, the process of monitoring the explicit effects of executing the action `(move r4 r5)` can deduce that the execution has failed because according to the self-localization module the robot is still in room `r4`. On the other hand, the process of monitoring the implicit expectations can declare that the room where the robot is located is a kitchen because an oven has been spotted in the current room.

In this chapter we use the second way where the process of monitoring the implicit expectations is called only when the explicit effects of the action are all verified in the world state produced by the execution of the current action. Hence, the final result of monitoring the execution of the action is the result returned by the process of monitoring its implicit expectations. Therefore, the schema of monitoring the execution of an action A in a world state ws includes the following steps:

1. The first step is a prediction step where the model of action A is used to compute the explicit effects of A when executed in ws.

2. In the second step, all the computed explicit effects are checked in the resulting world state. If all of them are verified, then the implicit effects of executing A in ws are derived and checked in the current world state

as well. In case one of the implicit expectations is found to be violated, a failure is returned to the plan executor. Otherwise, the execution of the action has succeeded and success is returned to the plan executor.

3. If in the second step, some of the explicit effects were found to be violated in the current world state, then a failure is returned to the plan executor.

4. The plan executor carries on the execution of the rest of the plan only if it gets "*success*" from the execution monitoring process. Otherwise a recovery procedure might be invoked to recover from the encountered unexpected situation.

4.3.2 Action Model

The predicted effects of an action are derived from its model, which is generally specified by an action template as part of the planning domain (see chapter 3, section 3.1). An action template represents a general schema that can be used to derive instances of actions by instantiating variables appearing as parameters of the action name. An action template has three parts:

1. An action name with a list of parameters.

2. A precondition part expressing under which conditions the action is applicable.

3. An effect part, i.e., what changes occur to the state where the action is applied.

In general, the effect part of the action contains two subsets of assertions about state variables. First, there is the subset of positive effects that contains state variables (fluents) asserted either to be true or to have a value that is different from false. The second subset contains state variables that are asserted to be false. It should be noted that there are other formalisms for describing action models. The interested reader is referred to the book by Ghallab and coauthors about the subject of automated planning [63]. For the purpose of this chapter, we will only consider monitoring the execution of deterministic actions,i.e, the models of such actions do not allow to specify uncertain effects. The execution of a deterministic action is assumed to produce the same effects every time it is applied in the same state.

Example The following template is used to specify the model an action with a deterministic outcome to enter a room whose name is to be bound to the variable `?r1`. The template is specified using the PTLplan first-order language used by the two planners PC-Shop and PTLplan, which are described in chapter 3.

```
(ptl-action
   :name    (enter ?r1)
   :precond (((?r0) (room ?r0)(robot-in = ?r0))
             ((?r1)(room ?r1)
                  (exists (?d)(door ?d)
                    (and (facing ?d)
                         (connects ?d ?r0 ?r1)(open ?d)))))
   :results (robot-in = ?r1))
```

The `:precond` part specifies the conditions under which the action is applicable. In this case it specifies that the robot is in another room `?r0` where it should be facing an open door `?d` that leads to the destination room `?r1`. The `:results` part specifies the predicted effects of the action. In this case, there is only one positive effect, i.e., `(robot-in = ?r1)`, which is used to assert that the robot will be in the destination room `?r1`. Notice that the predicate `robot-in` is in fact a state variable whose domain of values is the set of names of the different locations where the robot can be.

4.3.3 Components

Figure 4.3 shows the different components involved in the process of monitoring the execution of a plan action. The perception and anchoring component delivers perceptual information about objects. These objects are usually instances of the atomic classes defined in the semantic knowledge base, such as beds, chairs, etc. The perceptual information is computed using the data provided by the on-board sensors such as cameras and lasers. The information is represented in a symbolic format, and it consists of descriptions of objects in terms of their observed properties, e.g., color, shape, location, etc. For instance, if an object of type `sofa` has been perceived by the robot in room `r4`, the symbolic description is `(and (sofa sf1)(has-sofa r1 sf1))`. The perceptual information is used to estimate the current state of the world, which among other things includes the current estimated location of the robot. Perceptual information is also used to check the implicit expectations of the executed action.

The "robot plans/execution" component maintains information about the current plan and its execution context. It also stores planning domains that contain the formal descriptions of action models. The model-based predicted state of the world is computed using the model of the executed action and the previous world-state. The predicted state is compared against the estimated state to detect whether the explicit effects of the executed action were violated.

The knowledge representation and reasoning system LOOM is used to store and reason about semantic domain-knowledge. As explained in chapter 3.2, this knowledge is divided into terminological knowledge and assertional knowledge. The terminological knowledge captures general knowledge about the ob-

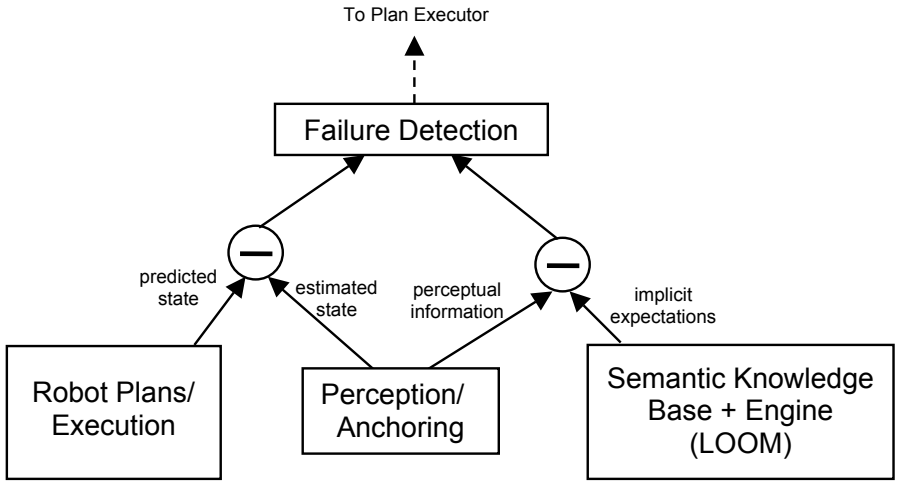


Figure 4.3: The different components involved in monitoring the implicit expectations of symbolic actions.

jects present in the environment of the robot. The assertional part of the knowledge base contains knowledge about individual objects that exist in the real world, such as which room is a kitchen and which object is the oven, etc. The semantic knowledge base is used to check the implicit expectations that should result from the execution of actions. Basically, the monitoring module asserts the acquired perceptual information about perceived objects that are related to the execution of an action. Then, the semantic knowledge base is queried to check whether an object of interest is automatically classified as an instance of a certain class.

4.4 Monitoring Implicit Expectations

A process for Semantic Knowledge-based Execution Monitoring, which we call ***SKEMon***, is outlined in figure 4.4 (This process will be referred to in subsequent chapters as crisp *SKEMon*). The process typically checks whether an execution-time object fits the description of an expected object *obj* using the available perceptual information. For instance, if the robot has executed the action `(pick-up c1)` to pick up the object identified by the symbol `c1`, then the object actually picked up by the robot is the execution-time object and needs to be checked to verify if it matches the description of `c1`, which is the expected object. Therefore, the *SKEMon* process is a kind of type checking process. If the robot has executed the navigation action `(enter r5)`, then the actual location where the robot has ended up is the execution-time object. The location where the robot is expected to be at, i.e., `r5` is the expected object whose de-

SKEMon(obj)

1. $CLs \longleftarrow$ SKB::get-asserted-classes(obj)
2. $temp \longleftarrow$ PERCEPTION::perceived-object
3. $\Pi \longleftarrow$ PERCEPTION::perceived-properties&relations($temp$)
4. SKB::create-instance ($temp, \Pi$)
5. **if** $\forall\ cl \in Cls$: SKB::is-instance-of($temp$, cl) **then**
6. return *success*
7. **else if** $\exists\ cl \in Cls$: SKB:is-not-instance-of($temp$, cl) **then**
8. return *failure*
9. **else**
10. return *unknown outcome*

End

Figure 4.4: Main steps of the semantic knowledge-based execution monitoring process **SKEMon**.

scription needs to match the actual location of the robot. In the pseudo-code of the process, the operations prefixed by "SKB::" involve using the semantic domain-knowledge, whereas those prefixed by "PERCEPTION::" involve using perceptual information.

The process gets as input the name of the expected object obj, which is derived from the action model. In our current implementation, obj is derived from the positive effects of the executed action. The process starts by querying LOOM about the asserted classes of the expected object obj (step 1). For example, if the robot executed the navigation action `(enter r4)`, then LOOM is asked about the asserted classes of the object named `r4`. Only the most specific asserted classes are considered, since the semantic knowledge base can deduce that an instance of a specific class is also an instance of all the more general classes. For instance, if `r4` is asserted once to be a room and once to be a living-room, then only the living-room class is considered.

In step 2, the execution-time object is given a temporary name, and in step 3 the monitoring process retrieves perceptual information about the perceived properties (as well as relations to the other perceived objects) of the execution-time object. It is worth mentioning that the perception and anchoring module retains only percepts that are relevant to the current domain by filtering the stream of percepts coming from the sensing modalities. The filtering is carried out by hard-coded functions that are defined as part of each domain. An alter-

native solution would be to automatically construct filters using the definitions of concepts and relations, which are stored in the semantic knowledge base.

The next step is to use the retrieved perceptual information to create a temporary instance referring to the execution time object in the semantic knowledge base (step 4). The aim of this step is to check whether the execution-time object can be automatically classified as an instance of one of the defined classes in the knowledge base. For instance, if the perceptual information indicates that the robot is in a room and that one chair `ch1` and one bed `b1` have been observed in that room, then the monitoring process asserts those facts in the semantic knowledge base by issuing the following LOOM command:

```
(tell (room temp)
      (has-chair temp ch1)
      (has-bed temp b1))
```

where `temp` is a temporary symbol used to refer to the current room (where the robot is actually located), i.e., the execution-time object. The execution of the command by LOOM results in an automatic classification of the newly created instance based on the properties and relations to the other perceived objects, i.e., the chair `ch1` and the bed `b1`.

Once the semantic knowledge base is done with the classification of the execution-time object, the monitoring process sends another query to LOOM to check whether the classification is consistent with the asserted classes of the expected object *obj* (step 5). In step 7, the monitoring process checks whether the available perceptual information reveals that one of the constraints, involved in the definition of the classes of the expected object, is violated. For our example, this is performed by sending the following two queries to LOOM:

```
(ask (living-room temp))
(ask (:not (living-room temp)))
```

The second query is asked only when the answer to the first one is "*NO*".
The monitoring process interprets LOOM's answers as follows:

- **Consistent Classification.** A *YES* on the first query means that the implicit expectations are verified and therefore the execution-time object *temp* is classified like the expected object *obj*. As a result, the *SKEMon* process returns *success* (step 6). In our example, this means that the robot is in the right type of room.
- **Inconsistent Classification.** A *YES* on the second query means that the classification of the execution-time object is inconsistent with the expected object *obj*. This occurs when at least one implicit expectation is violated. Hence, a *failure* is reported (step 8). In our example, this means that the robot is dislocated.

	m < n	m = n	m > n
`(:at-least n R)`	unknown	YES	YES
`(:exactly n R)`	unknown	unknown	NO
`(:at-most n R)`	unknown	unknown	NO

Table 4.1: Truth values of number constraints given as a function of `m`: the number of objects that have been observed to be related by relation `R` to a specific individual. The truth values are computed under the open-world assumption.

- **Unknown Outcome.** *NO* on both queries means that it cannot be determined whether some constraints (implicit expectations) hold or not (step 7). In our example, if no sofa is observed and the constraint `(:at-least 1 has-sofa)` is not known to be true or false for the current room, then the room cannot be classified as a living-room by LOOM.

The *unknown* outcome is due to the fact that we set up LOOM to operate according to open-world semantics. In other words, the facts told to LOOM are assumed to be only a part of the complete world state. As a result, LOOM assumes that there might be additional facts of which it has not been told. The reason behind using open-world semantics is to be able to take into account partial observability of the environment: due to occlusions, the robot gets only partial information about the presence of objects and their properties.

In fact, using open-world semantics makes number constraints prone to give an *unknown*. An `(:at-least n R)` constraint gives *unknown* whenever the total number of observed objects related to the constraint is less than the lower bound `n`. The constraint gives *YES* otherwise. An `(:at-most n R)` constraint gives *unknown* as long as the total number of observed objects related to the constraint is not above the upper bound `n`. The constraint gives *NO* otherwise. Table 4.1 shows the answers to queries about the truth value of three number constraints given as a function of the total number `m` of observed objects.

The robot has two options to handle the *unknown* outcome. The first option is to be credulous and consider the absence of counter-evidence as sufficient grounds for assuming that the execution of the action has succeeded. In our example, the credulous approach implies that the monitoring process should ask the semantic knowledge base whether the location is an instance of another class (that is not a superclass of the expected class) to check whether the robot is dislocated. If the class of the location is still not known, the monitoring process assumes that the location is correct as long as no evidence of the contrary is detected.

The second option is to take a cautious approach and actively try to gather more information in order to do a better classification. Chapter 6 presents in detail a sensor-based planning approach designed to deal with situations of lack of information in semantic knowledge-based execution monitoring.

4.5 Handling Unsuccessful Execution

Whenever the monitoring process finds out that an action has not been executed successfully, a recovery procedure can be launched to correct the unexpected situation. The recovery procedure consists of finding a sequence of actions that would lead to a situation where the robot can continue executing its top-level task plan. In our navigation example, replanning is needed when the robot is found to be dislocated. The first step in replanning is the creation of a world state that reflects the resulting unexpected situation, i.e., update the location of the robot to the right one. However, special care should be taken when performing location update because sometimes the new location might be not unique. For instance, if all what the robot has observed so far is a sink and sinks are defined to be either in a kitchen or a bathroom, then, the recovery module should take this fact into account.

4.6 An Illustrative Example

To clarify the idea of using semantic knowledge in monitoring the execution of symbolic plans, we describe an example of a mobile robot with manipulation capabilities. Test scenarios of a real robot acting in an indoor environment are presented in chapter 8.

In this example, the robot has an arm with a gripper, which is used to grab objects. The gripper has contact sensors whose states indicate whether the robot is holding some object. The robot is also supposed to have an on-board vision system that it can use to collect perceptual information about objects in its environment. The robot has a semantic knowledge-base that includes, among other concept and relation definitions, the following:

```
(defset shape :is
        (one-of 'cylindrical 'cubic 'spherical))

(defconcept bowl :is
            (and container
                 (:exactly 0 has-handle)
                 (= has-shape cylindrical))
(defconcept cup :is
            (and container
                 (:exactly 1 has-handle)
                 (= has-shape cylindrical))
```

Notice that the `defset` construct defines a concept as a set of symbols. In the definitions given above, the difference between a bowl and a cup is determined by the number of handles they can have.

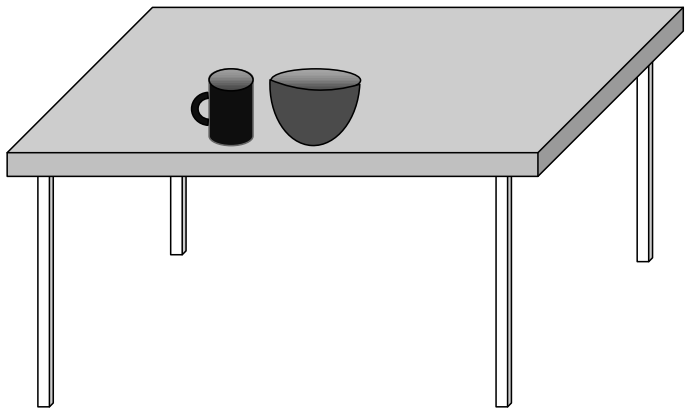


Figure 4.5: An example of a situation where the robot wants to pick up a bowl that happens to be placed next to a cup.

Suppose that the robot has just finished the execution of the plan action `(pick-up b1)` where `b1` is a symbol denoting an object of type `bowl`. The explicit effect of `(pick-up b1)` is to grab object `b1` which is encoded as `(holding b1 = t)`. Suppose also that instead of picking up the bowl, the robot picked up a nearby object of type cup (see figure 4.5).

Using the information provided by the contact sensors of the gripper allows the execution monitor to deduce that the explicit effect `(holding b1 = t)` holds. Therefore, the *SKEMon* process is called to check that the object held by the gripper is actually of type `bowl`. To do so, the *SKEMon* process gives a temporary name to the picked up object and then asks the perception process about the observed properties of the picked up object. Relying on the on-board vision system, the perception process replies that the object is a container with a cylindrical shape and that has one handle. This perceptual information is asserted by the *SKEMon* process in the semantic knowledge base as follows:

```
(tell (container temp)
      (has-handle temp h1)
      (has-shape temp cylindrical))
```

Next LOOM is asked about whether `temp` is of the same type as the expected object `b1`, i.e., whether `temp` is a bowl:

```
(ask (bowl temp))
```

The answer of LOOM to this question is *NO*, since the implicit expectation of a `temp` having no handle is known to be violated. As a result, the *SKEMon*

process concludes that the execution of the action has failed. One solution to respond to this unexpected situation is to put the cup on the table, then try to pick up the bowl. Notice that LOOM cannot classify the picked up object as a cup because it does not know whether that object has another handle.

4.7 Discussion

This chapter presented a novel high-level approach for monitoring the execution of symbolic task plans in indoor environments. The novelty of the approach resides in using semantic domain-knowledge to compute implicit expectations that are to be verified when plan actions are executed successfully. These implicit expectations are then checked based on the run-time collected perceptual information. In chapter 8 we describe test cases where the proposed approach was used on-board our mobile robots to monitor the execution of indoor navigation plans. We also describe an experimental methodology aiming at evaluating the performance of the proposed approach using simulation.

Although semantic knowledge plays an important role in many application areas, such as the semantic web [9] and image analysis and interpretation (see for example the work by Russ and colleagues [129] and the recent work by Neumann and Möller [112]), its use is still uncommon in mobile robotics. Notable exceptions include applications to facilitate human robot communication in a spoken language [142], classification of map spaces for navigation tasks [60], and as a means of publishing and sharing knowledge in multi-robot environments [32]. Simple geometric semantic knowledge has also been proved to be useful in building 3D maps of indoor environments [116], where simple semantic knowledge is expressed in terms of geometric constraints over laser readings to classify 3D points into floor, ceiling, or physical objects points. It is worth mentioning that the mobile robotics community has recently started looking at using semantic knowledge in robotics with more attention by organizing international events such as the workshop on semantic information in robotics at the IEEE international conference on robotics and automation [72] and the semantic robot vision challenge [118].

The process of monitoring the execution of symbolic plans needs to be performed at different levels, from detection of hardware errors to high-level deliberation. In this context, semantic knowledge can be one contributor to a multi-layered and multi-source monitoring process.

The use of semantic knowledge in execution monitoring helps detect failure situations that cannot be detected by traditional monitoring approaches. The reason is that in many situations, monitoring only explicit effects of actions cannot be enough for detecting unexpected situations. A direct consequence of using semantic knowledge in monitoring plans is that the process of planning to achieve tasks becomes less computationally demanding as the task planner does not reason about the details of the objects manipulated by the planning domain actions.

Although the proposed approach helps in developing more robust plan execution monitors, it still suffers from the inability to handle uncertainty in perception as well as actions with more than one possible outcome. In the next chapter, we present another approach of semantic knowledge-based execution-monitoring that aims to handle uncertainty in action effects, sensing and world states. Our aim is to develop an approach that is applicable to a wide variety of real world scenarios.

Chapter 5
Probabilistic Semantic Execution Monitoring

In chapter 4, we showed how semantic domain-knowledge can be used in the process of monitoring the execution of symbolic robot plans. The key idea is to use such knowledge to compute implicit expectations that can be observed at run time by the robot to make sure actions are executed correctly. The approach addressed actions with deterministic effects, i.e., having only one outcome and the result of evaluating the implicit expectations was treated in a boolean setting, that is either true, false, or unknown.

As uncertainty is an ever-present feature in mobile robotics, it has an impact on the actions of robots as well as their perception of their environment; in the presence of uncertainty, even the best laid plans can fail. Therefore, the main contribution of this chapter is the development of a second *SKEMon* approach that is be able to take into account uncertainty in world states, action effects, sensing, and the way expectations are interpreted in the semantic domain-knowledge. In this chapter, we describe a probabilistic quantitative-model of uncertainty, such that actions are allowed to have different outcomes each with a probability of occurrence and such that sensing can be unreliable. Using probabilities makes it possible to go beyond a boolean treatment of whether an expectation is verified. In particular, the execution monitor can combine different evidences in a systematic way. Therefore, given the a priori probabilities of the possible action outcomes, the available semantic knowledge, and the actual observations, the execution monitor is able to estimate the probability of whether a certain expectation is verified, such as "the robot is in an office with 0.9 probability" – in the framework of chapter 4, that would just have been a "unknown" monitoring result. Moreover, the fact that the a posteriori probability of each outcome of an action can be estimated enables a more informed decision about how to proceed (consider action execution successful, failed, or more information needed) than with just a boolean approach.

This chapter is organized as follows. In the next section, we give an overview of how semantic knowledge can be used under uncertainty. The rest of the chapter is devoted to explaining how uncertainty in sensing, action effects, and world states is incorporated in the monitoring process. Before concluding the chapter, we give an overview of how the results of execution monitoring are to be used by the plan executor in order to carry on the execution of the rest of the task-plan. Real-robot test scenarios showing the applicability of the approach as well as simulation experiments are reported in chapter 8.

5.1 Overview of the Approach

In chapter 3, we argued that the effects of uncertainty on plan execution can be mitigated by employing planning techniques that reason about uncertainty; the generated plans include therefore information collection actions that allow them to respond to different situations adequately. As those plans are generated off-line, the actual effects of their actions need to be estimated at execution-time, since the actual effects are not known beforehand. To estimate the actual effects of actions effectively, the monitoring process needs also to reason about the uncertainty inherent in action outcomes, world states, and sensing.

In the following, we give an overview of how semantic knowledge can be used in execution monitoring under uncertainty. The monitoring process uses the probabilistic model of actions, presented in chapter 3, to model actions with several possible outcomes. Thus, each outcome is associated with a prior probability. For instance, to model the uncertainty about the final result of moving from one room `?r1` to another connected room `?r2`, the specification of `(move r1 r2)` action might include two possible outcomes: either unintentionally remaining in `?r1` (e.g., due to wheel slippage and excessive turning) or effectively moving to room `?r2`. If prior probabilities for these two outcomes are available, then they can be specified as well, otherwise the two outcomes are assumed to be equiprobable. The template of the movement action can be specified using the PTLPLAN language as follows:

```
(ptl-action
   :name    (move ?r1 ?r2)
   :precond (((?r1)(room ?r1)(robot-in = ?r1))
             ((?r2)(room ?r2)
               (and (connected ?r1 ?r2)
                    (exists (?d)(door ?d)
                     (and (connects ?d ?r1 ?r2)(open ?d))))))
   :results (robot-in = (?r1 0.2)(?r2 0.8)))
```

The template specifies that the first outcome can occur with a probability that is equal to 0.2, while the second outcome can occur with a probability that is equal to 0.8.

This type of action templates can be used to find plans that reason about uncertainty in robot actions as well as sensing. This has the advantage of generating plans capable of dealing with potential contingencies that might arise at execution time (see section 3.1). For instance, the following plan could be generated to clean room `r2`, which is asserted to be of type `office`, starting from room `r4`, which is asserted to be of type `living-room`. The branches are created to handle the two predicted contingencies that might occur after the execution of the movement action[1]; the condition of each branch is the observation of the corresponding belief state (the `cond` form is used to introduce a conditional branching).

```
((move r4 r2)
   (cond ((robot-in = r4)
             (move r4 r2)
             (cond ((robot-in = r4) :fail)
                   ((robot-in = r2) (clean r2) :success)))
         ((robot-in = r2)
             (clean r2) :success)))
```

Even though the plan reasons about the possible contingency of not moving to the destination room, execution monitoring is still needed mainly for two reasons. First, the plan executor needs to know the outcome of the movement action in order to select the next action for execution. Second, other contingencies that were not planned for need to be detected should they occur at execution-time. For instance, the robot might not be initially in room `r4` which means that the execution of `(move r4 r2)` would result in a different situation other than being in room `r4` or room `r2`.

Typically, the monitoring process uses semantic domain-knowledge in the following way:

- For each possible outcome of the action whose execution is being monitored, a set of implicit expectations are computed. For instance, if one outcome is to be in an office, the implicit expectations of having at least one desk and at least one chair are computed. If the second outcome of the action is ending up in the printing-room, the implicit expectations would include having at least one printer, one copier, etc.

- Those expectations are used to estimate a probability distribution over what one would expect the actual world state to be like. For instance, the implicit expectation of seeing at least one desk implies that the probability

[1]Recall that a forward-chaining sensor-based planner starts with an initial belief state and adds actions until a belief state satisfying the goal is reached. If the insertion of an action gives rise to a set of belief states (due to partial observability), the planner inserts a branch for each resulting belief state and continues planning for each branch separately. Therefore, the resulting plan is conditional.

of having no desk is zero, while the probability of having one, two, or more desks is strictly greater than zero.[2]

Besides uncertainty about the world state, uncertainty in sensing is taken into consideration through a model that expresses the probability of what is observed for a given world state. In its general form, the sensing model permits:

- To state whether an object that exists in the real world is seen or not, e.g., to take occlusions into account.
- How a seen object is classified, i.e., the model accounts for misclassification of objects when they are seen. For instance, a sofa may sometimes be mistaken to be an arm-chair.

The monitoring process uses the prior probability distribution over the outcomes of the executed action together with the semantic knowledge-based probability estimates and the sensing model to compute the posterior probability of the outcomes. Thus, the monitoring task becomes more like a Bayesian belief update task [130]. This is basically done through computing the posterior probability distribution of the outcomes using the execution-time acquired information, i.e., the monitoring process computes the actual execution-time belief state.

More specifically, if $\mathbf{o}$ denotes the collected information (or observations hereafter), then the posterior probability of the action resulting in a specific outcome r is computed using Bayes formula:

$$p(r|\mathbf{o}) = \frac{p(\mathbf{o}|r)p(r)}{p(\mathbf{o})} \tag{5.1}$$

where $p(\mathbf{o})$ is a normalizing factor. The computation of the posterior $p(r|\mathbf{o})$ requires the specification of two probability functions: (1) the prior probability $p(r)$, and (2) the observation function $p(\mathbf{o}|r)$.

The function $p(r)$ is easily computed from the action model and the previous belief state bs, i.e., the belief state before the action was executed:

$$p(r) = \sum_{s \in bs} p(r|s)p(s) \tag{5.2}$$

where $p(r|s)$ is the conditional probability of the action resulting in outcome r when the world is in state s; $p(r|s)$ is specified in the action model as a conditional effect. Note that in the case of full observability, i.e., the previous belief state contains just one state, $p(r)$ is simply derived from the action model. Note also that using equation (5.2), we can easily take into account the output of probabilistic localization systems, such as particle-filter ones [56], as a belief

[2] As neither LOOM nor any other currently available DL system supports probabilities, the probability distributions of the expected state of the world are computed by a precoded procedure.

state: each state $s \in bs$ can represent a possible hypothesis about the true location of the robot.

Before going any further, we should mention that there are situations where equation (5.1) can result in an exception if $p(r|\mathbf{o})$ is zero for all outcomes r. These situations arise when the collected observations $\mathbf{o}$ constitute counter evidence against all outcomes, i.e., $p(\mathbf{o}|r) = 0$ for all outcomes r. An example of such situations is when the predicted outcomes of a movement action state that the robot can be either in room `r4` (living-room) or in room `r1` (a bedroom), but the robot sees a sink which is counter evidence against being in bedrooms or living-rooms. This indicates that the actual outcome is not one of the predicted ones.

One explanation of these exceptions is that the predicted outcomes were generated starting from a faulty model of the world, i.e., the world state used by the planner to instantiate the action is wrong. For instance, the robot is wrongly believed to be in room `r4` (the living-room) while it is actually in room `r5` (the kitchen).

Whenever the monitoring process encounters such exceptional situations, a recovery procedure is needed. The aim is to estimate a world state that is consistent with the acquired observations and then generate a new task-plan to achieve the assigned task. For our example, the new estimated world state would include all locations where seeing a sink is not counter evidence.

Deriving the Observation Function

In the following, we show how the observation function $p(\mathbf{o}|r)$ is computed. We will use the following notation: bold-face letters denote vectors, capitalized letters denote variables, and uncapitalized letters denote specific values of the variable denoted by the same letter but capitalized, e.g., o is the same as $O = o$ and $\mathbf{x}$ is the same as $\mathbf{X} = \mathbf{x}$. The i^{th} element of a vector $\mathbf{X}$ is denoted by X_i.

We will consider only observations that describe number constraints, i.e., specified by the `:at-least`, `:at-most`, and `:exactly` concept constructs in the semantic knowledge base. One could easily add random variables to represent the observation of other properties such as the color or the size of objects with a set of different values, and constraints over these values. For instance, we might have a constraint over the color of a specific class of objects with possible values in {`red, yellow, white`}.

To compute the observation function $p(\mathbf{o}|r)$, we need the following entities:

- A random variable R whose domain values $\{1, \ldots, m\}$ represent the different action outcomes that are specified in the action model. The prior probability $p(r)$ for each single outcome r is computed according to equation (5.2).

- A set of N atomic concepts defined in the semantic knowledge base. Each concept denotes a class of observable objects C_i. For example, $C_1 = \texttt{bed}$, $C_2 = \texttt{sofa}$, etc.
- A random vector $\mathbf{O}$ of size N such that its ith random variable O_i represents the number of observed objects of type C_i. For instance, if C_1 refers to the concept `bed`, O_1 represents the number of observed beds.
- A random vector $\mathbf{S}$ of size N such that its ith random variable S_i is a state variable whose values are the actual number of objects of type C_i. In our model, each state variable S_i depends directly only on R. Each state variable S_i takes values in a finite domain $V_i \subset \mathbb{N}$

Example Consider the execution of the navigation action `(move r4 r5)` whose model accounts for two possible outcomes. The first outcome, i.e., $R = 1$, is when the robot remains unintentionally in `r1`, while the second outcome, i.e., $R = 2$, is when the robot moves effectively to room `r5`. If the only classes of observable objects that can exist in either location are sofas and sinks, then S_1 and S_2 denote respectively the actual number of sofas and sinks that can exist in one of the rooms. O_1 and O_2 denote respectively the number of observed sofas and sinks in the current location.

We also assume that observing one object is independent of observing another; therefore number constraints are restricted to be rather over disjoint concepts. This means that we cannot have a number constraint about beds and at the same time another number constraint about big-beds, simply because observing an object that is of type bed is no longer independent of observing an object of type big-bed.

Thus, equation (5.1) becomes:

$$\begin{aligned} p(r|\mathbf{o}) &= \sum_{\mathbf{s}} p(r, \mathbf{s}|\mathbf{o}) \\ &= \alpha \sum_{\mathbf{s}} p(\mathbf{o}|\mathbf{s}) p(\mathbf{s}|r) p(r) \end{aligned} \tag{5.3}$$

where $\mathbf{s}$ ranges over values belonging to $V_1 \times V_2 \times \cdots \times V_N$, and $\alpha = 1/p(\mathbf{o})$ is a normalizing factor. To compute the observation function, two probability mass functions are required

- A sensing function $p(\mathbf{o}|\mathbf{s})$ that describes the probability of observing $\mathbf{o}$ when the real world state is described by $\mathbf{s}$.
- A state function $p(\mathbf{s}|r)$ that describes the probability of $\mathbf{s}$ when the outcome of the action is r. The computation of $p(\mathbf{s}|r)$ relies on the implicit expectations computed for outcome r using the available semantic domain-knowledge.

5.2 The Sensing Model

We start by deriving a general, but computationally expensive, sensing model; a simpler version is provided further down.

General Sensing Model

The function $p(\mathbf{o}|\mathbf{s})$ represents the sensing model of the robot. The term $p(\mathbf{o}|\mathbf{s})$ specifies that given certain actual values $\mathbf{s}$ for the state variables, what is the probability that we will observe o_1 objects of type C_1, o_2 objects of type C_2,$\cdots$, and o_N objects of type C_N? In its general form $p(\mathbf{o}|\mathbf{s})$ permits:

- To state if an object that exists in the real world is seen or not, e.g., thereby occlusions can be taken into account.
- How a seen object is classified, i.e., the model accounts for misclassification of objects when they are seen.

The potential for misclassifying objects when they are seen implies that all random variables in $\mathbf{O}$ and $\mathbf{S}$ depend on each other. Consequently, there is an exponential number of probabilities $p(\mathbf{o}|\mathbf{s})$ that need to be specified. We break this dependency by introducing N random vectors $\mathbf{G}_{i:1\leq i\leq N}$ (each of dimension $N+1$). Each $\mathbf{G}_i$ depends directly only on the ith state variable S_i, and $p(\mathbf{g}_i|s_i)$ expresses the probability of classifying s_i objects of type C_i as $g_{ik}(k=1\ldots N)$ objects of class C_k. The number of missed (unseen) objects of type C_i is denoted by $g_{i(N+1)}$. Figure 5.1 shows the dependency structure of the variables R, $\mathbf{S}$, $\mathbf{O}$, and $\mathbf{G_i}$.

Note that $o_i = \sum_{k=1,N} g_{ki}$, i.e., o_i represents the total number of objects classified as instances of C_i; either correctly, i.e., g_{ii} or incorrectly, i.e., g_{ki} (for $k \neq i$). For example, the number of observed chairs is the total number of objects classified (correctly or incorrectly) as chairs. Thus, we have

$$p(o_i|g_{1i},\ldots,g_{Ni}) = \begin{cases} 1 & \text{if } \sum_{k=1,N} g_{ki} = o_i \\ 0 & \text{otherwise} \end{cases}$$

Under the assumption of independently classifying observed objects of the same class, each $p(\mathbf{g}_i|s_i)$ can be represented by a probability mass function of a multinomial distribution whose parameters are $n = s_i$ and classification probabilities $p_1,\ldots,p_{N+1}$. The quantity p_k, for $1 \leq k \leq N$, represents the probability of classifying an object of type C_i as being of type C_k, while p_{N+1} is the probability of missing (not seeing) an object of type C_i.[3] Therefore, $p(\mathbf{g}_i|s_i)$ is given as follows:

[3]It is worth mentioning that most multi-class object classification methods allow to compute the probability of classifying an object under different classes (e.g., see the work by Mikolajczyk *et al.* [100] and the work by Sipe and Casasent [140]).

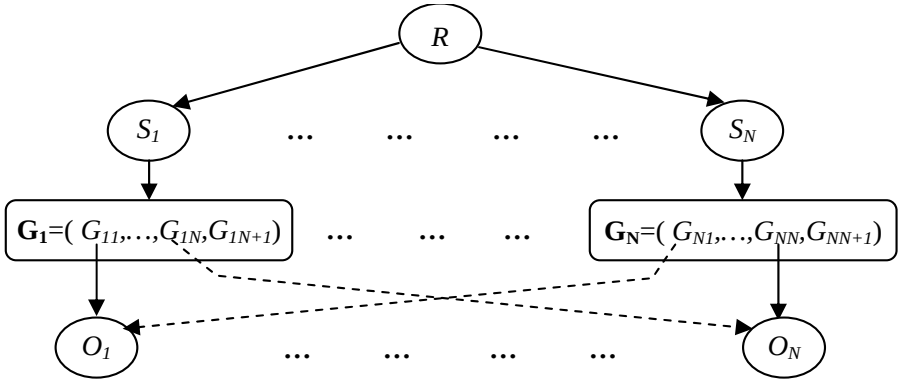


Figure 5.1: The dependency structure of the different random variables used in the state and sensing functions.

$$p(\mathbf{g}_i|s_i) = \begin{cases} \dfrac{s_i!}{\prod_{k=1}^{N+1} g_{ik}!} \prod_{k=1}^{N+1} p_k^{g_{ik}} & \text{when } \sum_{k=1}^{N+1} g_{ik} = s_i \\ \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

Finally, the general sensing model is formulated as:

$$\begin{aligned} p(\mathbf{o}|\mathbf{s}) &= \sum_{\mathbf{g_1},\ldots,\mathbf{g_N}} p(\mathbf{o}, \mathbf{g_1}, \ldots, \mathbf{g_N}|\mathbf{s}) \\ &= \sum_{\mathbf{g_1},\ldots,\mathbf{g_N}} p(\mathbf{o}|\mathbf{g_1}, \ldots, \mathbf{g_N}) p(\mathbf{g_1}, \ldots, \mathbf{g_N}|\mathbf{s}) \\ &= \sum_{\mathbf{g_1},\ldots,\mathbf{g_N}} \prod_{i=1}^{N} p(o_i|g_{1i}, \ldots, g_{Ni}) p(\mathbf{g_i}|s_i) \end{aligned} \tag{5.5}$$

The posterior probability of the outcomes given the observations is consequently given by

$$p(r|\mathbf{o}) = \alpha p(r) \sum_{\mathbf{s}} \prod_{j=1}^{N} p(\mathbf{s}_j|r) \sum_{\mathbf{g}_1,\ldots,\mathbf{g}_N} \prod_{i=1}^{N} p(\mathbf{o}_i|g_{1i}, \ldots, g_{Ni}) p(\mathbf{g}_i|\mathbf{s}_i) \tag{5.6}$$

It should be noted that the sensing model does not take into account hallucinations, i.e., seeing objects that do not exist at all; nevertheless hallucinations can be handled by a straightforward extension. TO do so, one can add a binary random variable H to indicate whether the robot is hallucinating, and

another random vector $\mathbf{G}$ that depends only on H. The values of $\mathbf{G}$ indicate the distribution of the hallucinated objects for the different classes.

Simplified Sensing Model

Because misclassification of observed objects has to be taken into account, the general sensing model in (5.5) can be quite expensive to compute. We also provide a simplified sensing model where objects can be missed (unseen) but not misclassified. In such cases, each observation random variable O_i becomes directly dependent only on its corresponding state variable S_i. Hence, we obtain:

$$p(\mathbf{o}|\mathbf{s}) = \prod_{i=1}^{N} p(o_i|s_i) \tag{5.7}$$

where $p(o_i|s_i)$ expresses the probability of seeing o_i objects of type C_i when there are in fact s_i of them. Under the assumption of independent observations of objects of the same type, the distribution of O_i given $S_i = s_i$ is a binomial $\mathrm{B}(n, p_i)$ with parameters $n = s_i$ and p_i is the probability of seeing an object of class C_i, i.e.,

$$p(o_i|s_i) = \binom{s_i}{o_i} p_i^{o_i} (1 - p_i)^{s_i - o_i} \tag{5.8}$$

5.3 Deriving the State Function

The state function $p(\mathbf{s}|r)$ is where semantic knowledge is encoded. This function gives for instance the probability that the grasped object has a handle given that it is a cup, or the probability that a room has a stove given that it is a kitchen. Unfortunately there is no workable description logic system that supports probabilistic reasoning (although some attempts have been made in that direction, e.g., see the work by Koller *et al.* [86] and the recent work by Lukasiewicz [94]). Therefore, the probabilities of the state functions are implemented outside the semantic knowledge base.

As we consider that each state variable S_j is dependent only on the outcome of the action R, the state function $p(\mathbf{s}|r)$ becomes

$$p(\mathbf{s}|r) = \prod_{j=1}^{N} p(s_j|r) \tag{5.9}$$

Each $p(s_j|r)$ specifies the probability of having exactly s_j objects of type C_j given that the outcome of the action is known to be r. We use semantic knowledge as a basis for computing $p(s_j|r)$. The key idea to determine implicit expectations $E_r = \{e_1, \ldots, e_{n_r}\}$ for each outcome r. Each expectation expresses a number constraint over the values of the actual number of objects of a certain type C_j, i.e., $e_j \equiv (S_j \in V_j)$ where $V_j \subset \mathbb{N}$.

For instance, using the model of the action `move` given above, to instantiate the action `(move r1 r4)` to move from the bedroom `r1` to the living-room `r4` gives two outcomes. The first outcome is when the robot is still in `r1`, and the second one is when the robot moves into `r4`. Since `r1` is a bedroom, and since bedrooms are defined in the semantic knowledge base as rooms having at least one bed, at most one sofa, and no sink, the implicit expectations of the first outcome could be $E_1 = \{e_1, e_2, e_3\}$ where $e_1 \equiv (S_1 \in \{1, 2, \ldots, m\})$, $e_2 \equiv (S_2 \in \{0, 1\})$, and $e_3 \equiv (S_3 \in \{0\})$. S_1, S_2, and S_3 are random variables describing respectively the number of beds, sofas, and sinks that may be in `r1`. The value of m specifies the maximum number of beds that can be in a bedroom.

Once the implicit expectations related to each outcome are computed, they are used to specify $p(s_j|r)$ as follows:

- If there is an implicit expectation $e_j \in E_r$ constraining the values of a state variable S_j, i.e., $e_j \equiv (S_j \in V_j)$, we should have

$$\begin{array}{ll} p(s_j|r) = 0 & \text{if} \quad s_j \notin V_j \\ 0 < p(s_j|r) \leq 1 & \text{if} \quad s_j \in V_j \\ \sum_{s_j \in V_j} p(s_j\ |r) = 1 & \end{array}$$

  The probability mass function $p(s_j|r)$ can be a known mass function used in counting processes such as the binomial or the Poisson mass functions. It can also be given as a table of probabilities reflecting the belief of the user.

- For those state variables S_j that are unconstrained in r, we simply assume that $S_j \in \{0, 1, \ldots, \max(S_j)\}$, where $\max(S_j)$ is a predefined upper bound on the values of S_j, and take $p(s_j|r)$ to be a uniform probability mass function.

As stated above, the user can use a known probability mass function to model the state function $p(s_j|r)$. This can be considered as a systematic way of specifying $p(s_j|r)$, when the user does not have access to a table of estimated values of the state function. Using a known probability mass function reduces considerably the amount of information that the user has to provide, since only a few parameters need to be specified. For instance, the shifted geometric mass function can be used to model a state function representing an `:at-least` expectation $e_j \equiv$ `(:at-least n R)` as follows:

$$P(S_j = m|r) = \lambda(1 - \lambda)^{m-n} \quad \text{for} \quad m \geq n \tag{5.10}$$

where λ is a parameter that specifies the probability of having exactly one object of type C_j. If the user wants to impose a maximum n_{max} on the values that the random variable S_j can take, a modified version of (5.10) can be used as specified in equation (5.11).

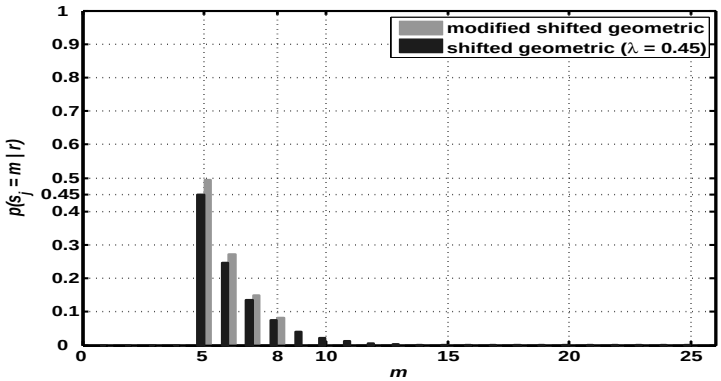


Figure 5.2: Bar-plots of a shifted geometric mass function (with $\lambda = 0.45$ and $m \geq 5$) and its modified version with $n_{max} = 8$.

$$P(S_j = m|r) = \frac{\lambda(1-\lambda)^{m-n}}{\alpha} \quad \text{for} \quad n \leq m \leq n_{max} \tag{5.11}$$

where α is a normalizing factor that is equal to $\sum_{m=n}^{n_{max}} \lambda(1-\lambda)^{m-n}$. Figure 5.2 shows an example of a state function modeled with a shifted geometric mass function with $\lambda = 0.45$ and $n = 5$ as well as its modified version with $n_{max} = 8$.

Example Suppose that the robot has executed the movement action `(move r1 r4)` to move from bedroom `r1` to room `r4`, which is a living-room. Deriving the state function for the first outcome implies first deriving the implicit expectations implied by being in `r1`. Suppose that the concept of `bedroom` defined in the semantic knowledge base gives the following implicit expectations: $e_1 \equiv$ `(:at-least 1 has-bed)`, $e_2 \equiv$ `(:at-most 1 has-sofa)`, and $e_3 \equiv$ `(:exactly 0 has-sink)`.

Consequently, the conditional probabilities of the state variables given that the robot is in a bedroom, might be determined as follows:

- The number of beds in a bedroom can be modeled as a shifted geometric random variable S_1, i.e.,

$$P(S_1 = i|r) = \lambda(1-\lambda)^{i-1} \quad \text{if} \quad i \geq 1$$

where λ is the probability of having exactly one bed. Using a shifted geometric probability mass function ensures that the probability decreases when the number of beds increases. For instance, when $\lambda = 0.8$, we get $P(S_1 = 1|r) = 0.8 < P(S_1 = 2|r) = 0.16 < P(S_1 = 3|r) = 0.032 < \cdots$

- As sofas are believed to be uncommon in a bedroom, one might use a probability mass function that reflects this belief, for instance:

$$P(S_2 = 0|r) = 0.8; P(S_2 = 1|r) = 0.2$$

  where S_2 is a random variable modeling the number of sofas.
- The implicit expectation e_3 implies that the random variable S_3, representing the number of sinks, has as probability mass function:

$$P(S_3 = 0|r) = 1.0$$

- As there is no implicit expectation about chairs in a bedroom (case 2 above), we can have the uniform probability mass function:

$$P(S_4 = 0|r) = \cdots = P(S_4 = 4|r) = \frac{1}{5}$$

  which means that there might be up to 4 chairs in a bedroom.

Deriving the state functions for the second outcome, i.e., being in the living room, is done the same way.

Once the execution monitor finishes computing the posterior probabilities of the outcomes of the executed action, the plan executor needs to be notified about the result of the execution. The plan executor uses that result to continue the execution of the rest of the task-plan or to try to recover if needed. In the next section, we explain how this is done.

5.4 Using the Results of Execution Monitoring

The results of monitoring the execution of an action are computed based on the type of the plan being executed and the assumptions made by the planner to generate it. In other words, the execution monitor needs to know whether the planner has generated the task plan taking into account partial observability as well as uncertainty about action effects and sensing.

5.4.1 Linear Plans

If the task plan was generated under the assumptions that there is no uncertainty about the state of the world and that actions are deterministic, then the plan executor expects to be notified about whether the execution of an action has resulted in "success" or "failure". The result "success" reflects the situation where the actual outcome produced by the execution of the action is the same as the outcome predicted by the planner. On the other hand, the result "failure" reflects situations where the predicted outcome is different from the actual

outcome. Naturally, the plan executor carries on the execution of the rest of the task plan only if it gets 'success", otherwise a recovery procedure might be needed to respond to the resulting execution failure.

As we have seen so far in this chapter, the execution monitoring process is designed to reason about the different forms of execution-time uncertainty, regardless of whether the planner does the same. Since the result of execution monitoring is a probability distribution over the possible outcomes of the executed action, the computed probabilistic posterior needs to be mapped to the binary result expected by the plan executor. One way of achieving this mapping is to return "success" whenever the outcome with the highest posterior probability is the same as the predicted outcome. More formally, let r denote the outcome predicted by the planner and r' the outcome with the highest posterior probability of the executed action, i.e.,

$$r' = \underset{R=i}{\operatorname{argmax}} P(R = i|\mathbf{O} = \mathbf{o}) \tag{5.12}$$

then the execution result to return to the plan executor is computed as follows:

$$\begin{array}{llll} \text{If} & r = r' & \text{return} & success \\ \text{otherwise} & & \text{return} & failure \end{array}$$

One can also use a modified selection criterion that uses the value of a threshold T to determine the resulting outcome r':

$$\begin{array}{llll} \text{If} & r = r' \text{and} P(R = r'|\mathbf{O} = \mathbf{o}) \geq T & \text{return} & success \\ \text{otherwise} & & \text{return} & failure \end{array}$$

where r' is computed according to equation (5.12). This modified criterion can be used to avoid returning "success" when the highest posterior probability might be low. For instance, if an action might result in four possible outcomes, then it is possible to have a situation where the outcome with the highest posterior has a probability of 0.26, which might be considered to be very low to conclude that the execution of the action has succeeded.

One issue that might arise when using the modified criterion is that there might be no outcome that satisfies it. To cope with such an issue, one can either fall back to the criterion in equation (5.12) or try to identify which of the resulting outcomes would satisfy the selection criterion. This is usually achieved by actively gathering information that is capable of reducing the uncertainty surrounding the resulting situation. Section 6.6 of chapter 6 presents an information gathering solution that is designed to such purpose.

Example Suppose that the robot is in the living-room (room `r4`) and that the next action to execute is `(move r4 r5)` to move to the kitchen (room `r5`). Suppose that the planner used a deterministic model of the action, i.e., its predicted outcome is that the robot will be in room `r5` with certainty. The

non-determinism of the action is handled by the monitoring process by using the probabilistic model with two outcomes as given in section 5.1 above. This means that the first outcome, $R = 1$, models the situation of the robot getting stuck in the initial room `r4` with $P(R = 1) = 0.2$, while the second outcome is when the robot effectively moves to the destination room `r5` with $P(R = 2) = 0.8$.

If the posterior computed by the monitoring process is $P(R = 1) = 0.4$ and $P(R = 2) = 0.6$, and the threshold T is equal to 0.5, then the result of execution is "success". However, if $T = 0.7$, the monitoring process will not be able to compute r', as neither outcome has a posterior that is greater than 0.7. In this case, the monitoring process can decide to launch an active information gathering procedure to look for evidence (or counter evidence) of being in one of the rooms. On the other hand, a fall back to (5.12) will result in "success".

5.4.2 Conditional Plans

In case the planner reasons about uncertainty in action outcomes and the initial conditions of world states, the generated task plans are conditional; each plan branch is associated with one possible modeled action-outcome (see chapter 3). Therefore, the executor of the conditional task-plan expects from the monitor not a "success" or "failure" result, but the actual outcome of the executed action. As a result, the plan executor selects and executes the branch associated with the actual resulting outcome. This means that an execution failure is raised, if the resulting outcome has no branch associated with it, e.g., because it was not predicted by the planner to occur. Consequently, a recovery procedure might be needed to continue the execution of the task plan.

As monitoring yields a probability distribution over possible outcomes, the monitoring process needs to select one of them to return to the plan executor. One way of doing that is to return the outcome with the highest posterior probability as specified in equation (5.12). Another way is to use the modified threshold criterion together with information gathering in case there is no outcome that fulfills the selection criterion.

5.4.3 Conditional Plans under Partial Observability

If the task plan was generated taking into account partial observability of the environment, the plan executor needs to know the actual belief state resulting from action execution, to be able to continue the execution of the plan. As probabilistic monitoring is equivalent to computing the actual belief state of the world, the whole posterior probability distribution is returned to the plan executor. The plan executor selects the plan branch associated with the belief state whose observations are consistent with the run-time collected ones and

whose probability distribution is equal to the computed posterior. Otherwise, an execution failure is raised, which leads to starting a recovery procedure.

5.5 Summary and Conclusions

This chapter presented a semantic knowledge-based execution monitoring approach that is designed to handle uncertainty in action effects, sensing and world states. The approach provides more informed decision about how to proceed after an action is executed (execution can succeed, fail, or more information is needed) than with the boolean approach presented in chapter 4.

In summary, semantic knowledge is used to compute implicit expectations for each possible action outcome. Then, a probability distribution is estimated to reflect how the world state looks like taking into account those implicit expectations. Moreover, a probabilistic model is provided to reason about uncertainty in sensing for a given world state. Thus, the result of monitoring is a posterior probability distribution over the different action outcomes.

The use of probabilities to model uncertainty in sensing and acting gives us a well founded treatment, but providing the needed probability values might be a daunting task for large domains. In our implementation, we took a Bayesian approach and interpreted probability values as measures of belief. Therefore, the user can model probability state functions using known probability mass functions such as the shifted geometric and Poisson distributions. Using a known probability mass function reduces considerably the amount of information that the user has to provide, since only a few parameters need to be specified. We also simplified the task of providing conditional probabilities for the sensing model by making assumptions that allowed us to use well known probability mass functions (binomial and multinomial functions) to encode the probability of seeing objects as well as misclassifying them when they are seen.

Unfortunately there is no workable description logic system that supports probabilistic reasoning in both world descriptions and general domain-knowledge (although some attempts have been made in that direction [86]). Therefore, we were obliged to implement the probabilistic model of the world state outside the semantic knowledge base. In fact, research on reasoning under uncertainty with description logics is an on ongoing activity [94, 95].

In our probabilistic sensing model, observations are restricted to belong to different branches of the taxonomy, i.e., no observation belongs to a superclass of another observation. This is a limitation that would be circumvented by using new developments in reasoning with uncertainty at all levels of the taxonomy of concepts. Moreover, the sensing model is relatively complex, and it may be expensive to compute for large state spaces. We have proposed a simplified model that assumes no misclassification. Alternatively, one may use approximate inference methods to address the computational complexity of the sensing model. This last point is worth investigating further and therefore constitutes a subject of future work.

Chapter 6
Information Gathering for Monitoring

The crisp *SKEMon* process, which was presented in chapter 4, used the immediately available perceptual information to evaluate implicit expectations of executing actions correctly. The result of the evaluation permits to determine whether the expectations (1) are confirmed, (2) are violated, or (3) it cannot be determined whether they hold or not, e.g., due to that only parts of the location or objects under observation can be seen at the moment. Case 3 brings us to the problem we tackle in this chapter, i.e., expectations are not always immediately observable.

In this chapter, we extend the crisp *SKEMon* developed in chapter 4 to handle situations involving lack of information. We present an active information-gathering schema for modeling and reasoning about uncertainty due to incomplete information. We show how such a schema is used to collect the required information that would help evaluate the implicit expectations with unknown truth values. Although the treatment of uncertainty does not include probabilities, chapter 7 shows how the same schema uses probabilities to reason and recover from situations of anchoring failures that are caused by ambiguity in perceptual information.

In this chapter, we use sensor-based planning to generate the necessary actions that the robot can execute in order to reduce or eliminate uncertainty that is due to incomplete information. Our choice is motivated by the fact that sensor-based planning has the ability to handle complex situations involving lack of information in an effective and automatic way. The key idea is to model the occurring situation as a planning task under partial observability. In this regard, the initial state of the planning task represents the situation of incomplete information, while the goal represents a situation where that information is available. Therefore, the generated plan includes movement and observation actions needed to gather the required information. We also present a greedy

strategy to handle situations involving lack of information in the probabilistic version of *SKEMon*, where information that is likely to reduce the uncertainty in the action outcome is selected to be collected.

This chapter is organized as follows. In the next section, we describe a scenario where information gathering is needed for execution monitoring. Then, we describe how sensor-based planning can be used to find active information gathering plans, and how those plans are executed and monitored. Before concluding the chapter, we describe the greedy strategy for identifying information to collect for the purpose of probabilistic *SKEMon*.

6.1 A Motivating Scenario

Let us reconsider the scenario of the home environment given in chapter 4. Suppose that the robot is in the kitchen (room `r5`) and that it has a scheduled task to clean the living-room (room `r4`) because few guests are coming later in the evening. To achieve the assigned task the on-board planner generates a plan that could include the following actions: `((goto d4);(enter r4);(clean r4))`. Now suppose that the robot has just finished the execution of the `(enter r4)` action, and that the monitoring process is triggered to check the resulting outcome. Assuming that the self-localization module indicates that the robot is in room `r4` (i.e., the explicit effect of the action is verified), then the crisp *SKEMon* process is launched to check the implicit expectations of being in a living-room.

Suppose that all what the robot could perceive from its current place is a chair and a table. As chairs and tables can be in rooms of different types, the acquired perceptual information is not enough to help the *SKEMon* process to establish whether the current room is a living-room. Therefore, the initial result of monitoring is "unknown". At this stage, the *SKEMon* process can assume a credulous approach and consider that the robot is in room `r4`, since it could not perceive any counter evidence against being in a living-room. However, if the robot wants to be sure that the task is achieved successfully (e.g., the robot does not want to end up cleaning a room that is not the desired one), it needs to gather more information that helps it establish that it is in the right room. The information to look for is needed to conclude whether the implicit expectations of being in the living-room are verified or violated. For example, the robot might look for sinks to check that the implicit expectation of having no sinks in a living-room is not violated. It can also look for sofas as a living-room is supposed to have at least one sofa.

6.2 Planning to Gather Information

The situation encountered by the robot in the example above represents an unexpected situation that is due to lack of information, which is needed to determine whether the robot executed its action successfully. In other words,

it could not be determined whether some implicit expectations hold or not. In the following, we assume a cautious approach, where the robot has to look for information required to evaluate such expectations to establish whether an action has been successfully executed.

We propose an active information-gathering approach that is based on automatically analyzing and encoding the unexpected situation as a planning problem under partial observability. We use sensor-based planning to solve the problem at hand; the solution is a plan that includes actions designed to collect information capable of helping the robot establish if implicit expectations hold or not. The procedure of handling a situation involving lack of information using planning can be summarized as follows:

1. Situation assessment: first the resulting situation of lack of information is analyzed with the aim of creating a planning problem through the specification of an initial belief state and a goal to reach. The initial belief state encodes the resulting situation as a set of hypotheses. Each hypothesis represents one possible assignment of truth values to the implicit expectations that are currently with unknown truth values. Situation assessment also involves determining the conditions under which the needed information can be acquired, e.g., the locations where the robot can see more objects. In addition, the initial belief state contains other information useful for conducting planning, such as topological information. On the other hand, the goal of planning is to be able to know whether the implicit expectations hold or not.

2. Plan generation: the created planning problem is, then, passed to the sensor-based planner to solve it by generating a plan containing movement and information-gathering actions. The movement actions are meant to put the robot in a state where it can execute the information gathering actions. For instance, moving to a location where it is possible to observe other parts of a room. Another example of a movement action is rotating an object held by the gripper of the robot in order to observe an initially hidden side of that object.

3. Plan execution and monitoring: the penultimate step consists in launching the execution of the information-gathering plan. To do so, each action is translated into a set of sensorimotoric processes that are executed by the low-level control architecture of the robot (e.g., in our case it is the ThinkingCap architecture). Moreover, the same monitoring framework is used to monitor the execution of the information-gathering plan. This means that situations of recursive planning are possible. This happens when the monitoring of an action returns "unknown" as a result of execution; therefore another information-gathering plan needs to be generated. It is worth mentioning that there is no risk of infinite recursive

monitoring as description logics do not allow definitions of concepts to be cyclic.

4. Resuming the execution of the task plan: depending on the result of the execution of the last action of the information-gathering plan, the implicit expectations can be concluded to hold or not. If all the implicit expectations were found to hold in the real world, then the execution of the task plan is resumed. However, if there was at least one implicit expectation that was found to be violated, then the monitoring process informs the plan executor that the execution of the action of the task plan had failed. This means that a recovery episode is needed aiming at handling the execution-failure situation either by finding another plan to accomplish the initial task or by repairing the failing task-plan (see chapter 2 for a information about recovery strategies).

   In case the execution of the information-gathering plan itself fails (e.g., the robot cannot reach an observation location due to obstacles), the plan executor can try to launch a recovery procedure that produces another information-gathering plan or it can fall back to the credulous monitoring approach and use the collected perceptual information to deduce whether the execution of the last action of the task plan has succeeded.

The generation and execution of information-gathering plans requires information from different sources. First, we obviously need the semantic knowledge base in order to generate the implicit expectations. Second, a planning domain is needed, specifying among other things observation actions designed to collect information and establish the truth values of expectations. Other types of information include spatial information such as topological maps and the spatial relations between objects. In the following we explain how the planning domain is modeled. The planning domain is primarily designed to handle situations involving lack of information detected by the crisp *SKEMon* framework. In section 6.6 we give an overview of information gathering for the purpose of probabilistic *SKEMon*.

6.3 Modeling the Planning Domain

The planning domain contains action templates that can be instantiated and synthesized to solve planning problems. Our solutions are based on using the sensor-based planners PTLplan and PC-Shop (see chapter 3) to generate active information-gathering plans. In the following, we describe the different types of actions used by both planners. We also describe methods used by PC-Shop to decompose abstract tasks into more detailed ones.

6.3.1 Actions

There are two types of actions that can be used to deal with situations of lack of information that is needed to know whether implicit expectations hold or not. First, actuation actions are needed to put the robot in a position where it can collect information. Examples of such actions are approaching a cup to be able to determine its content or moving to a location inside a room to have a better view of its currently hidden parts. Second, observation actions are needed to collect the missing information. This can be, for example, smelling the content of the cup to determine whether it contains a specific substance. We have a number of different actions for different types of expectations. Since the implicit expectations to evaluate are specified by atomic concept constructs, the planning domain contains an observation action for each atomic concept construct. The aim of each observation action is to gather information so that the related concept construct can be evaluated to hold or not.

Actions for atomic concepts An observation action `(check-<A> ?x)` is associated with each observable atomic concept `A`. The effect of such an action is to collect information required to verify whether the object, identified by the symbol bound to `?x`, is an instance of `A`. Thus, `(check-room r10)` checks if the object denoted by `r10` is an instance of the atomic concept `room`, e.g., by checking the map constructed by the robot at execution-time. Similarly, action `(check-container c1)` checks whether object `c1` is an instance of the atomic concept `container`.

Actions for number and type constraints To establish the truth value of expectations specified by number and type constraint constructs (i.e., `(:at-least n R)`, `(:at-most n R)`, `(:exactly n R)`, and `(:all R C)`), we need to define an action template to handle each constraint construct. The aim of such an action is to keep track of the observed objects that can be fillers of role `R`; such information is used to deduce the truth value of the corresponding constraint. For instance, action `(eval-all ?r ?c ?x)` keeps track of the perceived objects that are related to `?x` by `?r`. It concludes that `(:all ?r ?c)` is verified only if those individuals are *all* of type `?c`.

Here is a detailed description, in the PTLplan language, of the action template `(eval-at-least ?n ?r ?x)`, which is part of a navigation domain. This template is associated with the `:at-least` atomic concept construct. The objective of this action is to check whether the number constraint `(:at-least ?n ?r)` is verified. To do so, its execution consists in looking for objects related to the object instance `?x` by relation `?r`. For example, `(eval-at-least 2 has-bed r1)` is meant to look for objects of type `bed` to check whether room `r1` has at least 2 beds.

```
(ptl-action
 :name    (eval-at-least ?n ?r ?x )
 :precond (((?p)(place ?p)(robot-at = ?p))
           ((?r)(role ?r)(and (not (checked ?r ?p))
                              (can-check ?r ?p)))
           ((?x)(room ?x)(part-of ?p = ?x)
                         (not (nec (at-least ?n ?r ?x)))))
 :results (and (checked ?r ?p = t)
           (cond ((at-least ?n ?r ?x)
                  (obs (at-least ?n ?r ?x = t)) )
                 ((and (at-least ?n ?r ?x = f)
                   (forall(?l)(can-check ?r ?l)(checked ?r ?l)))
                  (obs (at-least ?n ?r ?x = f)))
                 ((true)
                  (and (obs (at-least  ?n ?r ?x = f))
                       (at-least ?n ?r ?x = t f))))  ))
```

The intuition behind the action is that the robot can move between different places, and at each position it can observe a number of objects related to `?x` by `?r`. While doing that, it keeps track of the total number of observed objects and compares it to `?n`.

The initially false predicate `(checked ?r ?l)` denotes whether the robot tried to observe objects, needed to evaluate the relation `?r`, from position `?l`. The initial truth value of the predicate `(at-least ?n ?r ?x )` is unknown. The action is intended to observe the truth value of this predicate to determine if the constraint `(:at-least ?n ?r)` is true for the object bound to `?x`. The modal formula `(nec` α`)` is used to denote that the amodal formula α is true in all the possible worlds of the belief state where the action is applied.

In short, the precondition part specifies when the action is applicable. In this case, the truth value of `(at-least ?n ?r ?x)` is not known and the robot is at a location where it is possible to observe individuals related to `?x` by relation `?r`. The result part encodes the effects of the action. Besides asserting `(checked ?r ?l = t)`, the action has also three conditional outcomes specified with the `cond` form. Note that the `cond` form works essentially like the Lisp `cond`, and the `obs` form is used to encode run-time observations.

1. The first outcome is observing that the constraint is verified. This happens when the predicate `(at-least ?n ?r ?x)` is true, i.e., at execution time, at least `?n` objects have so far been observed to be related to `?x` by `?r`.

2. The second outcome is observing that the constraint is violated. This is the result when the robot has visited all locations where it is possible to perceive objects that satisfy `?r`, yet their total number is still less than `?n`.

3. Finally, if the total number of perceived objects is less than ?n and there are locations where the robot may observe extra objects, then the third outcome is to observe that (at-least ?n ?r ?x) is not verified and asserting its truth value to be unknown.

Actions associated with the other atomic concept constructs (:at-most, :exactly, etc.) are defined in the similar ways. The definitions of all the actions are given in appendix A.2.

6.3.2 PC-SHOP Methods

Besides actions that correspond to primitive tasks, PC-SHOP uses methods to specify how to decompose abstract tasks into subtasks. In other words, methods are procedures that provide the knowledge of how to control the process of searching for plans (refer to chapter 3, for more information on planning with PC-SHOP). In the context of handling situations involving lack of information, the interest is in determining the truth value of expectations with unknown truth values. A method is provided to specify how to collect information to determine the truth value for each type of expectation. In the following, we discuss the method associated with the :at-least expectation in a navigation domain. This should give a clear idea of how to write methods for the other types of expectations.

```
(method (!check-at-least ?n ?r ?x)
      /* Alternative 1*/
     (((?p)(place ?p)
       (and (finish = f)(robot-at = ?p)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (:immediate eval-at-least ?n ?r ?x)
               (:cond ((at-least ?n ?r ?x)
                       (:immediate !eval-termination))
                      ((not (at-least ?n ?r ?x))
                       (!check-at-least ?n ?r ?x))))
     /* Alternative 2*/
     (((?p)(place ?p)(and (finish = f)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (move ?p)(!check-at-least ?n ?r ?x))

     /* Alternative 3*/
     (true)
     (!eval-termination) )
```

Briefly, the method (!check-at-least ?n ?r ?x) describes how to achieve the task of determining the truth value of (:at-least ?n ?r) expectations

for an object instance `?x`. The method defines a recursive process of active information gathering with three alternative expansions:

- The first alternative captures the situation when the robot is at place where it is possible to look for objects that can fill the relation `?r`. The expansion specifies that the primitive task `(eval-at-least ?n ?r ?x)` is to be executed immediately [1] which results in a conditional branching. The first branch reflects the case where the expectation `(:at-least ?n ?r ?x)` is verified, which implies that no more information gathering is required. Therefore, the next task to execute is to check whether planning should be terminated. In the second branch `(at-least ?n ?r ?x)` is not verified. Consequently the robot needs to keep looking for fillers of `?r` from other positions. Thus, a recursive call to `(!check-at-least ?n ?r ?x)` is inserted.

- The aim of the second alternative is to force the robot to move to places where it is possible to look for objects that can fill the relation `?r`. The tasks of the second alternative are expanded only when the precondition of the first alternative is not verified. Moreover, there has to be a place `?p` where the robot has not yet looked for fillers of `?r`. The expansion specifies that the robot should move to place `?p` and then recursively execute the task `(!check-at-least ?n ?r ?x)`.

- The third alternative is expanded only when the first two alternatives are not applicable. The only thing to do in this case is to execute the task `(!eval-termination)` to check whether planning should be terminated.

Notice that planning using the method `(!check-at-least ?n ?r ?x)` stops as soon as the corresponding expectation is found to be verified, i.e., at least `?n` objects have been observed to be related to `?i` by `?r`. Therefore, going around to look for more objects is avoided. The idea behind the methods `(!check-at-most ?n ?r ?x)` and `(!check-exactly ?n ?r ?x)` is similar, except that planning continues to look for fillers of `?r` from all places before concluding that the corresponding expectation holds.

In situations where there are several expectations to check, we need to make sure to stop planning as soon as it is needed. For instance, if we are checking a conjunct of expectations, we need to stop planning once one of them is violated. On the other hand, if we are checking a disjunct of expectations, planning should be terminated, as soon as one of the expectations is known to be verified. The method `(!eval-termination)` is used to do just that:

[1] Since primitive tasks are executable actions, their specification is the same as PTLplan action templates with the same name. The only exception is that primitive tasks do not have preconditions.

```
(method (!eval-termination)
      /* Alternative 1*/
     ((()(nec planning-formula)))
      (stop success)
      /* Alternative 2*/
      ((()(nec (not planning-formula))))
      (stop violated)
     /* Alternative 3*/
     (true)
     (:nop))
```

where `planning-formula` is a formula that relates all the expectations we need to check. For instance, if the goal is to determine whether the room where the robot is located is a bedroom, the `planning-formula` is given as a conjunct of the expectations of being in a bedroom:

```
(and (room r1) (at-least 1 has-bed r1)(at-most 1 has-sofa r1))
```

The primitive task `(stop success)`, respectively `(stop violated)`, is meant to stop the planning process once `planning-formula` is satisfied, respectively violated. The result of the corresponding action is visible through setting the value of the `(finish)` fluent to true, i.e.,

```
(ptl-action
   :name (stop ?res)
   :precond ()
   :results (and (finish = t)(result = ?res)) )
```

6.4 Planning Process

PTLplan requires the specification of an initial belief state and a goal formula as input. To use PC-Shop, one needs to provide an initial belief state and an initial list of tasks to accomplish.

6.4.1 Initial Belief State

Generating plans to collect information successfully implies that the planner takes into account the issue of partial observability of the environment. To this end, belief states are used to represent the robot's incomplete and uncertain knowledge about the world at some point in time, i.e., a belief state represents a set of hypotheses about the actual state of the world given past observations.

The initial belief state contains hypotheses about the truth value of each expectation to check. This is done by asserting that the expectations can be true

or false. The initial belief state includes also other information needed for the planning task, e.g., the robot's whereabouts, knowledge about the workspace, etc.

Example Suppose that the robot has just finished the execution of the action `(enter r1)` where `r1` is an instance of `bedroom`, which is defined as follows:

```
(defconcept bedroom :is
           (:and room
                (:at-least 1 has-bed)
                (:at-most  1 has-sofa)))
```

Suppose that the robot could establish that its final location is a room, that is the implicit expectation `(room r1)` is verified. Suppose also that the robot has not seen any sofa nor any bed yet inside the room where it is. This means that the implicit expectations corresponding to the constraints `(:at-least 1 has-bed)` and `(:at-most 1 has-sofa)` are not known to hold or to be violated. This situation is encoded as follows:

```
(and (room r1)(at-least 1 has-bed r1 = t f)
              (at-most  1 has-sofa r1 = t f))
```

where "`= t f`" means the truth value can be either `t`(rue) or `f`(alse). The formula encodes a belief state with four hypotheses (possible worlds) that result from the different combinations of the truth values of the expectations. Notice that the symbol `r1` in this situations is a temporary symbol that refers to the current location of the robot, which might be different from the expected location.

The process in charge of creating the initial belief state needs to incorporate other information that is needed to carry out the planning task. This includes a symbolic description of the topological map of the robot environment as well as the places where it is likely to observe individual objects or features related to an implicit expectation with unknown. For instance, the robot can decide that in order to look for beds inside the room where it is located, it needs to scan it from two places `r1-1` and `r1-4`. Therefore, it adds `(can-check has-bed r1-1)`, `(part-of r1-1 = r1)`, etc., to its belief state.

6.4.2 Goal Specification

The goal state to be reached by the plan is specified by a modal formula containing a conjunct of the predicates associated with the expectations whose truth values are unknown. The predicates are assigned the expected truth value of their corresponding expectations. For the previous example, the goal formula to pass to PTLplan is:

```
(nec (and (at-least 1 has-bed  r1 = t)
          (at-most  1 has-sofa r1 = t)))
```

which expresses that the information-gathering plan should try to reach a state where it is known that room `r1` has at least one bed and at most one sofa.

6.4.3 Initial Tasks of PC-SHOP

The list of initial tasks of PC-SHOP includes a task for each expectation with unknown truth value. The planning termination condition, i.e., `planning-formula` needs to be set at this stage as well. For the previous example, the initial set of tasks is specified as

```
(:unordered (!check-at-least 1 has-bed r1)
            (!check-at-most  1 has-sofa r1))
```

while `planning-formula` is given as

```
(and (at-least 1 has-bed r1 = t)
     (at-most  1 has-sofa r1 = t))
```

6.4.4 Plan Generation

Both planners PTLPLAN and PC-SHOP were used to generate the information-gathering plans. PTLPLAN takes as input an initial belief state and a goal formula, while PC-SHOP starts with the initial belief state and the list of initial tasks. Both planners generate conditional plans that when applied in the initial belief state they lead to a belief state where the goal formula is verified.

To be able to resume the execution of the top-level task-plan, information-gathering plans are restricted to include only actions that do not alter the state of the top-level task-plan in any relevant way. For example, the information-gathering plan to verify the implicit expectations of being in room `r1` is not allowed to include actions to move outside `r1`. In navigation scenarios that involve only observation and movement actions and where the top-level actions are used only to move to a certain room, such a restriction is sufficient. A more flexible approach is to require that certain conditions hold at the end of the execution of the information-gathering plan, such as the robot being in the same room where the monitoring process triggered the information-gathering procedure. This would make it possible to include actions to go outside that room to acquire some information that is not possible to acquire otherwise.

Example The following plan is generated by PTLPLAN for checking whether `r1` is a bedroom, starting from the situation where the truth values of the implicit expectations `(at-least 1 has-bed r1)` and `(at-most 1 has-sofa r1)` are unknown.

```
((eval-at-least 1 has-bed r1)
 (cond ((at-least 1 has-bed r1 = f)
        (move r1-2)(eval-at-most 1 has-sofa r1)
        (cond ((at-most 1 has-sofa r1 = t)
               (move r1-4)(eval-at-most 1 has-sofa r1)
               (cond ((at-most 1 has-sofa r1 = t)
                      (eval-at-least 1 has-bed r1)
                      (cond ((at-least 1 has-bed r1 = f)
                             :fail)
                            ((at-least 1 has-bed r1 = t)
                             :success))
                      ((at-most 1 has-sofa r1 = f) :fail)))
              ((at-most 1 has-sofa r1 = f) :fail)))
        ((at-least 1 has-bed r1 = t)
         (move r1-2)
         (eval-at-most 1 has-sofa r1 )
         (cond ((at-most 1 has-sofa r1 = t)
                (move r1-4) (eval-at-most 1 has-sofa r1)
                (cond ((at-most 1 has-sofa r1 = t) :success)
                      ((at-most 1 has-sofa r1 = f) :fail)))
               ((at-most 1 has-sofa r1 = f) :fail))))))
```

The plan includes movement actions that cause the robot to move to observation places (`r1-1`, `r1-2`, and `r1-4`) inside the current room as well as observation actions aimed at evaluating the truth values of the expectations. The special action `:fail`, respectively `:success`, denotes predicted failure, respectively predicted success in satisfying the goal. Notice that the plan declares failure as soon as the observation `(at-most 1 has-sofa r1)` evaluates to false, meaning that more than one sofa has been seen in `r1`.

Note that the fact that the plan includes movement and observation actions is specific to navigation scenarios and not a restriction. In other scenarios, movement actions might not be needed, but observation actions will always be necessary, since the aim is to gather information. For example, if the robot is executing the action `(grab c21)`, where the symbol `c21` refers to a cup that contains coffee, the observation plan would include actions to check the content of the cup but no actions to change the location of the robot.

The monitoring module concludes that the implicit expectations are verified when the last executed action of the information-gathering plan is `:success`. Reaching the `:fail` action implies that there was at least one violated expectation.

6.5 Plan Execution

The execution of the information-gathering plan is carried out by the plan executor described in chapter 3. Basically, the execution of a plan action is handled by a set of low-level executable actions called xactions. Each xaction defines a procedure that calls the functions of the ThinkingCap robot control architecture (see section 3.3.1) to produce fuzzy behaviors to achieve a low-level goal.

Example To execute the high-level action `(move r1-2)` to move to place `r1-2` inside room `r1`, the associated xaction procedure calls the navigation behavior-planner B-Planner (part of the ThinkingCap) to achieve the low-level goal specified by the fuzzy predicate `(at me r1-2)`. The generated B-Plan contains a set of fuzzy behaviors that are executed in parallel by the low-level controller. Moreover, the execution of the B-Plan is monitored by a process implemented by a hand-coded procedure. The following B-Plan is generated to achieve the fuzzy goal `(at me r1-2)` starting from a second place inside the same room `r1`:

```
IF (at me r1-2)                THEN  still(goal)
IF ((near me r1-2) and
    (not (at me r1-2)))        THEN  reach(r1-2)
IF ((at me r1) and
    (oriented me r1-2) and
    (not (near me r1-2)))      THEN  reach(r1-2)
```

where the conditions of the rules are fuzzy-logic formulas that define the context of each rule while the consequents are fuzzy behaviors. The degree to which each rule is active depends on how much the current state of the world satisfies the condition of the rule.

The xactions associated with high-level observation actions are meant to control the robot sensors to acquire perceptual information about the environment. Typical perceptual information includes the color, the shape, and the relative position of objects detected by the on-board vision system. This perceptual information is used to assert facts needed to evaluate the truth value of the predicate corresponding to the concept construct. For instance, executing the `(eval-at-most 1 has-sofa r1)` action results in adding information needed to evaluate the truth value of `(at-most 1 has-sofa r1)`, that is, all newly perceived sofas are added to the execution-time belief state. In other words, the assertion `(and (sofa sf) (has-sofa r1 sf))` is executed for each newly perceived sofa `sf`.

The evaluation of branching conditions is performed by a procedure that queries the execution-time belief state about the truth values of the predicates

forming the condition. In particular, the actual truth value of an observation predicate associated with a number constraint can be computed by comparing the number of observed objects, filling the corresponding relation, with the cardinality specified in the constraint. Figure 6.1 shows a procedure designed to evaluate predicates associated with number constraints (i.e., `(at-most n R i)`, `(at-least n R i)`, and `(exactly n R i)`) and type constraints (i.e., `(some R C i)` and `(all R C i)`). The procedure gets as input an observation predicate *obs* (e.g., `(at-most 1 has-sofa r1 = t)`) and the execution-time belief state *bs*. The procedure outputs the truth value "true", if *obs* holds in *bs*. Otherwise, the truth value "false" is returned. The output is determined in step 12 by evaluating the logical equivalence of the expected truth value (i.e., the one specified in *obs* and computed in step 4) and the actual truth value (i.e., the one computed using the execution-time belief state *bs* in steps 7–11) (step 12).

evaluate-constraint(obs, bs)

1. $R \longleftarrow$ relation(obs)
2. $i \longleftarrow$ instance(obs)
3. $n \longleftarrow$ cardinality(obs)
   $C \longleftarrow$ type(obs)
4. $v \longleftarrow$ truth-value(obs)
5. $O \longleftarrow \{o \mid bs \models R(i, o)\}$
6. **case** C
7. **at-least:** $res \longleftarrow (|O| \geq n)$
8. **at-most:** $res \longleftarrow (|O| \leq n)$
9. **exactly:** $res \longleftarrow (|O| = n)$
10. **some:** $res \longleftarrow (bs \models \exists o \in O.\ C(o))$
11. **all:** $res \longleftarrow (bs \models \forall o \in O.\ C(o))$
12. **return** $(res \Leftrightarrow v)$

END

Figure 6.1: A procedure for evaluating whether observation predicates associated with number or type constraints hold in the execution-time belief state.

For example, applying the procedure in figure 6.1 to check if the observation predicate `(at-most 1 has-sofa r1 = t)` holds in the current belief state relies on computing the number of sofas that have been observed inside `r1`. If that number is equal to 1 (which is the cardinality specified in the predicate), then "true" is returned. However, if `(at-most 1 has-sofa r1 = f)` is the predicate to check, then "false" is returned. This result is due to the fact that the expected truth value of the constraint, i.e., "f" (for false) is not the same as the actual one, i.e., "true".

The execution of the actions of the information-gathering plan are also monitored by the same monitoring process used to monitor the task plan. This

means that a new information-gathering plan might be needed to monitor the execution of an observation action which results in recursive active monitoring. Note that there is no risk for infinite recursive information-gathering, as description logics do not allow cyclic definitions of concepts.

6.6 Information Gathering for Probabilistic *SKEMon*

In chapter 5, we discussed situations where the collected observations are not enough to compute a posterior that matches what was predicted by the task-planner. For instance, the planner might have predicted that after the execution of a "move" action, the robot would be either in the living-room `r4` or in the kitchen `r5`; thus, if the robot does not see anything from its current location, then the monitoring process will not be able to tell with precision in which room the robot is. As discussed in section 5.4, the probabilistic *SKEMon* process can select the outcome with the highest posterior probability as the resulting outcome. We also mentioned that the outcome with the highest posterior probability that exceeds a given threshold T can also be selected; however, we argued that there can be no outcome that satisfies the second criterion, due to the high uncertainty in the computed posterior of the action outcomes.

As in the case of crisp *SKEMon*, the robot can collect information useful to reach a situation where uncertainty in action outcomes is eliminated or at least reduced. In this section, we describe how such information can be identified without planning all the way to a state where all the uncertainty is eliminated. The key idea is to select the information that is likely to achieve the highest reduction in uncertainty and then plan to collect it. In section 8.1.4 of chapter 8, we describe a test scenario where information gathering is applied to reduce the uncertainty about the posterior of a navigation action.

One measure that can be used to select information to gather is information gain, which expresses the average reduction in the uncertainty in a random variable. More formally, the information gain $IG(R|O_i)$ achieved for the random variable R, denoting the possible action outcomes, when the values of an observation variable O_i are known is given as follows:

$$IG(R|O_i) = H(R) - H(R|O_i) \tag{6.1}$$

where the quantity $H(R)$ is called the information entropy of the random variable R and $H(R|O_i)$ is called the conditional entropy of R given O_i. The quantity $H(R)$ is used to measure uncertainty in a discrete random variable R. It is defined as follows:

$$H(R) = -\sum_{R=r} p(r)\log(p(r)) \tag{6.2}$$

The conditional entropy $H(R|O_i)$ represents the average uncertainty in R taking into account that the value of O_i is known, i.e.,

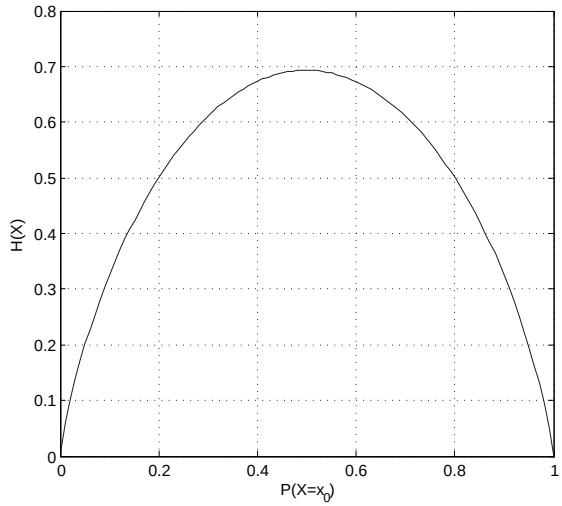


Figure 6.2: Plot of the entropy $H(X)$ of a binary random variable X with outcomes in $\{x_0, x_1\}$. The plot shows the amount of uncertainty in X for all possible probability distributions of X. It is clear that the highest uncertainty is attained when X is uniformly distributed.

$$H(R|O_i) = \sum_{O_i=o_i} p(o_i)H(R|O_i = o_i) \tag{6.3}$$

It is worth mentioning that information entropy was introduced by Shannon in 1948 [137] as a measure to quantify the information contained in a message composed of a finite set of symbols. The units for entropy are "nats" when the natural logarithm is used and "bits" for base 2 logarithms. Larger values of $H(R)$ represent higher uncertainty in R. In fact, $H(R)$ attains its maximum value when R is uniformly distributed (see figure 6.2).

For the purpose of information gathering to reduce uncertainty about action outcomes, we need to compute the information gain taking into account all observation variables. In other words, equation (6.1) is applied to all observation random variables O_i, and the observation variable that gives the highest information gain is then selected as the useful information to collect. In other words, let obs be the observation to collect, then obs is determined as follows:

$$obs = \underset{O_i}{\operatorname{argmax}}\, IG(R|O_i) \tag{6.4}$$

Once $obs = O_k$ is determined, the robot can start looking for objects that are instances of class C_k. For instance, if equation (6.4) reveals that $obs = O_1$,

which denotes the number of observed sinks, then the robot can move to an observation place where it can scan the current room to look for sinks.

One obvious limitation of using equation (6.4) as the selection criterion is that it does not take into account the cost inherent in collecting information. This is not an issue when all observation actions have the same cost, e.g., due to using the same sensing modality. However, there are scenarios where not all missing information has the same cost. For instance, it is known that acquiring odour information by an electronic nose can take longer times (from 1 to 3 minutes) than acquiring visual information [91]. Another example of cost is power consumption needed to accomplish an information-gathering task.

One can opt for a decision-theoretic approach to take into account the cost of information gathering the same way it is performed in active sensing strategies (Hager and Mintz [68], Burgard *et al.* [28]). To do so, a utility function U needs to be defined. For instance, Fox *et al* [55] define the utility associated with a sensing action as the decrease in uncertainty about the robot location, where uncertainty is measured by entropy. The cost of each sensing action is then subtracted from its utility, and the action that has the highest expected utility is selected. In our case, we would need to associate a utility with each observation random variable O_i. This can be measured by the information gain function specified in equation (6.1). As a result, the observation that achieves the highest difference between its utility and its cost is selected as the information to gather. In other words, given the utility $U(O_i)$ and costs $C(O_i)$ of all observations O_i, the robot selects the observation obs as follows:

$$obs = \underset{O_i}{\operatorname{argmax}}(U(O_i) - C(O_i)) \tag{6.5}$$

where $U(O_i) = IG(R|O_i)$.

Once the information to collect is identified, a plan can be generated to collect it, e.g., by moving to different places in a room to scan parts that were initially hidden. The advantage of planning to collect only a piece of information at a time is that the planning problem becomes much less complex. However, using a greedy approach does not guarantee the computation of an optimal solution.

6.7 Discussion

The main focus of this chapter has been on using sensor-based planning to handle situations where the acquired information is not enough to know whether the execution of an action has succeeded or failed. The key idea is to model the resulting situation involving lack of information as a multi-hypothesis planning problem where some information needs to be gathered in order to evaluate expectations with uncertain truth values. We also discussed a greedy approach for selecting information to collect based on using information-theoretic measures.

Several research works have investigated the use of planning for different tasks that involve information gathering. For instance, Kovacic *et al.* [87] have used planning to collect visual information for 3D object recognition. Koenig and Liu [85] used information-gathering actions as part of path plans with the aim to avoid taking difficult paths, e.g., containing muddy sections. Similarly, information gathering is used to achieve safer robot navigation by Miura and Shirai [102] and Gancet and Lacroix [61]. However, no work has addressed using sensor-based planning to collect information for the purpose of monitoring the execution of plans.

The advantage of using planning is the flexibility of dealing with a multitude of unpredictable and complex situations of incomplete information. In other words, only one planning domain needs to be specified to automatically compute solutions for a multitude of situations of lack of information. Moreover, by combining semantic domain-knowledge and sensor-based planning, the proposed approach achieves an interleaving of planning and execution, which is, according to Nourbakhsh and Genesereth [113], a desired ability of autonomous robotic systems acting in uncertain environments. In fact, the task planner can reason on a more abstract level (office, kitchen, etc.) to generate the task plan, while the monitoring process takes care of checking the details (desk, oven, etc.) at execution-time. The monitoring process, in turn, uses sensor-based planning when it is needed to do so.

A clear issue that might arise when using planning in general is that planning is a computationally demanding process especially if the planner has to reason about uncertainty and partial observability of the environment. We also consider the issue of which expectations should be selected for checking as an important one, as the number of expectations might be very large. One possible solution to this issue is to use a criterion that permits to select only important expectations, e.g., by using the information gain measure to identify which constraints need to be checked first.

Chapter 7
Handling Anchoring Failures

So far, the focus of this thesis has been on the process of monitoring the effects of actions and its important role in achieving correct execution of robot task-plans. In this chapter, we address another class of plan execution failures that are detected by the anchoring process. Anchoring is the problem of establishing the correspondence between a symbol and a percept. A type of anchoring failure is when the robot cannot determine which percept corresponds to a given symbol due to ambiguity, i.e, there are several plausible percepts. To be able to execute plan actions successfully, the robot needs to solve such problematic situations.

Generally, ambiguous situations in anchoring arise when the robot does not have access to all the perceptual information that is necessary to identify the correct object to anchor. This is another case of unexpected situations that are due to lack of information. Therefore, it can be seen as an application of the information-gathering schema described in chapter 6. In other words, to recover from such a failure, the robot creates a description of the current situation and generates a plan that disambiguates that situation.

While the sensor-based planning approach described in chapter 6 treated all hypotheses equally when creating the initial belief state, in this chapter hypotheses are treated in a probabilistic way. This makes it possible to assign probabilities to what perceived object should be anchored to the given symbol. Moreover, information-gathering plans can be generated so that the current situation can be disambiguated with a threshold probability. Another similarity to chapter 6 is the use of background knowledge to fill in missing information about perceived objects. The difference is that the background knowledge is provided in terms of general probabilistic first-order rules rather than as a description logic knowledge-base. Although the work presented in this chapter does not enrich the use of semantic domain-knowledge in monitoring the execution of plans, it is still considered to be important for the process plan execution, as the final goal of the work of this thesis is robust execution of robot

plans. The chapter builds upon the initial work of Broxvall *et al.* [27] and its related works by the author and others [83, 22].

This chapter is organized as follows. In the next two sections we review the concept of perceptual anchoring and anchoring failures due to ambiguity. A detailed description of how to recover from ambiguous situations in anchoring is given in the subsequent section. The chapter includes also test scenarios performed to show the applicability of the approach.

7.1 Overview of Perceptual Anchoring

Anchoring is the process of establishing and maintaining the correspondence between the perceptual data and symbolic abstract representation that refer to the same physical object [35]. Intelligent embedded systems using symbolic representations, such as mobile robots, need to perform some form of anchoring in order to achieve their tasks.

Consider a mobile robot, equipped with a vision system and a symbolic AI planner, trying to find dangerous gas bottles in a building on fire. Suppose that the planner has generated a plan that contains the action `(go-near b1)`, where the symbol `b1` refers to an object described as "a green gas bottle". The `go-near` action is implemented by a sensori-motor loop that controls the robot using the position parameters extracted from a region in the camera image. In order to execute the plan action `(go-near b1)`, the robot must make sure that the vision percept it is approaching is the one of the object identified by the symbol `b1`. Thus, the robot uses a functionality called **Find** to link the symbol `b1` to a region in the image that matches the description "a green gas bottle". The output of **Find** is an anchor that contains, among other properties, the current position of the gas bottle. While the robot is moving, a functionality called **Track** is used to update this position using new perceptual data. Should the gas bottle go out of view for some time the **Reacquire** functionality would be called to update the anchor as soon as the gas bottle is in view again [37].

7.1.1 Matching

An important process in all the anchoring functionalities is the matching between the symbolic description $Desc(o)$, of the object of interest o, given by the planner and the attributes of percepts generated by the sensor system. A percept π consists of information about an object derived from sensor data (e.g., a video image), such as estimates of shape and color. These estimates are often uncertain, e.g., due to noise in sensing or lack of perceptual information.

The matching process decides which percepts to use to create or update the anchor of a given symbol. The result of matching $Desc(o)$ against the properties of π can be either *no* match, a *partial*, or a *complete* match [36]. A percept π is said to be completely matching o if all the properties of $Desc(o)$ match those of π, and partially matching if it at least a property of the description

$Desc(o)$ cannot be determined to match its counterpart in π due to uncertainty. A percept is said to be a complete anchoring candidate for a symbol o if it fully matches $Desc(o)$, and a partial anchoring candidate if it partially matches $Desc(o)$.

7.1.2 Relational Properties

There are situations when it is relevant to describe objects not just in terms of their properties but also their relations to other objects. By considering relations, we may be able to resolve cases where the known properties of the object are not sufficient to distinguish it from other similar objects. An example is "the green garbage can that is *near* the red ball and the blue box". The object that needs to be anchored, in the example "the green can", is considered the *primary object* while the other objects related to it, in the example "the red ball" and "the blue box", are *secondary objects* [27]. In our work, we use in particular binary relations and we allow for descriptions to have several nested relations. In practice, the depth of the relational description is always limited.

Definition 1 *Let O denote the set of object symbols. A relational description of an object $o \in O$ having m binary relations ($R_{k;1\leq k\leq m}$) with m secondary objects ($o_{k;1\leq k\leq m}$) is denoted $Rel_{desc}(o)$ and it is defined recursively as:*

$$Rel_{desc}(o) =_{def} Desc(o) \bigcup_{1\leq k\leq m} \{R_k(o, o_k)\} \cup Rel_{desc}(o_k)$$

Obviously, the relational description of any object o includes its properties (specified in $Desc(o)$), the binary relations where o is involved ($\bigcup_k \{R_k(o, o_k)\}$)[1] as well as the relational descriptions of all the k secondary objects o_k in relation with o ($\bigcup_k \{Rel_{desc}(o_k)\}$).

Example The description "the red ball that is near the blue box that is on the brown table" refers to a primary object o_1 "the red ball", a secondary object o_2 "the blue box", and a third object o_3 that is secondary to o_2. The relational descriptions of the three objects are given as:

$$\begin{aligned}
Rel_{desc}(o_3) &= \{(\texttt{shape}\ o_3 = \texttt{table}), (\texttt{color}\ o_3 = \texttt{brown})\} \\
Rel_{desc}(o_2) &= \{(\texttt{shape}\ o_2 = \texttt{box}), (\texttt{color}\ o_2 = \texttt{blue})\} \cup \\
&\quad \{(\texttt{on}\ o_2\ o_3)\} \cup Rel_{desc}(o_3) \\
Rel_{desc}(o_1) &= \{(\texttt{shape}\ o_1 = \texttt{ball}), (\texttt{color}\ o_1 = \texttt{red})\} \cup \\
&\quad \{(\texttt{near}\ o_1\ o_2)\} \cup Rel_{desc}(o_2)
\end{aligned}$$

[1] In case the second argument of the relation is the primary object symbol, we can always rename the relation such that the primary object symbol is the first argument

The anchoring process handles relational descriptions by considering relations as additional properties of the object to anchor, i.e., $\{R_k(o, o_k)\} \cup Rel_{desc}(o_k)$ is an additional "complex" property of object o. In the previous example, the fact that o_1 is *near* "a blue box that is on the brown table" is an additional complex property of o_1 besides being a ball that has the color red. Clearly, a relational property has the additional complexity that an anchor needs to be found also for the secondary object. The anchoring process for the secondary objects is the same as the one for the primary object: the secondary object can be described as definite or indefinite and it can have complete, partial or no anchoring candidates.

In practice, all possible candidates for the primary object o are considered based on the non-relational properties in its description, i.e., $Desc(o)$. Then, for each of these candidates, anchors for all secondary objects are looked for on the basis of their descriptions and their relations to the primary object.

Definition 2 *A relational anchoring candidate for an object o having m binary relations ($R_{k;1\leq k\leq m}$) with m secondary objects ($o_{k;1\leq k\leq m}$) is represented by a list $(\pi_0, (\pi_{11} \ldots), (\pi_{21} \ldots), \ldots, (\pi_{m,1} \ldots))$ such that π_0 is a candidate percept for the primary object o, and for each secondary object o_k, a (possibly empty) list $(\pi_{k,1} \ldots)$ of all candidate percepts satisfying $Rel_{desc}(o_k)$ and relation $R_k(o, o_k)$.*

Notice that the same definition applies recursively to relational anchoring candidates for secondary objects. In fact, a relational anchoring candidate can be easily represented using an `and-or` tree where the `and` nodes represent the relations and the `or` nodes represent the candidate percepts satisfying the relation of the parent node.

The process of matching a relation description $Desc(o)$, of an object o, against perceptual information produces a set of relational anchoring candidates where:

- A relational anchoring candidate is *completely matching* if π_0 completely matches $Desc(o)$ (the primary object) and for each secondary object o_k there is only one (definite case), or at least one (indefinite case) candidate percept π_{kj} completely matching $Desc(o_k)$. The definite/indefinite distinction here refers to whether the secondary object symbol is definite/indefinite.
- A relational anchoring candidate is *partially matching* if for some object (including the primary one) there is no completely matching percept.

7.2 Failures and Ambiguities in Anchoring

There are situations where the anchoring module cannot create or maintain an anchor for a specific symbol from its percepts because of the presence of ambiguity, i.e., it cannot determine what anchoring candidate to choose. The

	# Matches		Definite		Indefinite	
Case	full	partial	result	action	result	action
1	0	0	Fail	Search	Fail	Search
2	0	1+	Fail	Observe	Fail	Observe
3	1	0	Ok	—	Ok	—
4	1	1+	Ok/Fail	-/Observe	Ok	—
5	2+	any	Conflict	—	Ok	—

Table 7.1: The different situations that can occur during anchoring.

anchoring module detects the presence of ambiguity on the basis of the number of complete and partial anchoring candidates. Another important factor is whether the symbolic description $Desc(o)$, of the object to anchor o, is making a definite reference to exactly one object in the world (e.g., "the red ball") or an indefinite reference, i.e., to any matching object (e.g., "a red ball"). Table 7.1 summarizes the different situations that can occur [27].

The situation of having no anchoring candidates that fully match the symbolic description are represented by cases 1 and 2. Recovery in case 1 can be achieved by looking for the desired object in another location. In case 2, an information-gathering plan can be generated and executed to go around partial candidates with the aim of observing the missing properties of the object. The successful disambiguation of the occurring situation makes it possible for the anchoring module to identify which of the candidate perceived objects should be used for anchoring.

In case 3, the anchoring module selects the percept that was found to fully match the description of the desired object.

Cases 4 and 5 reflect situations where at least one complete candidate for the symbol is perceived. Therefore, if the symbolic description is indefinite, any one of the complete candidates can be selected to anchor the symbol. When the description is definite, case 4 can be handled either by observing all partial candidates to eliminated them, or the complete candidate could be selected. In case 5, the situation cannot be recovered from by collecting more perceptual information as the matchings are full; thus, unless the description can be made more constrained, the situation is considered to be unrecoverable.

Example The robot is presented with the symbolic description "`g1` is a garbage can near a red ball with a mark" and given the task to go near `g1`. To do this, the robot needs to anchor the symbol `g1`. Consider a situation where a garbage can π_0 and a red ball π_{11} are perceived, but no mark is visible on the ball. In this situation, we have a singleton set $\{(\pi_0, (\pi_{11}))\}$ of relational candidates. There is one fully matching percept π_0 for the primary object and one partial match π_{11} for the secondary object; the mark was not observed. Consequently the entire relational candidate is a

partial match, giving rise to ambiguity. Thus, to be sure that the observed garbage can is the requested one, the red ball has to be checked for marks from other viewpoints.

Dealing with Ambiguous Situations

When the robot is not able to execute a plan action due to ambiguity in anchoring objects, a recovery procedure is needed to compute a solution to handle the recoverable cases outlined above. Typically, the solution includes information-gathering actions that allow the robot to establish whether certain properties of the candidate objects hold or not. Therefore, the schema used to cope with ambiguous situations in anchoring resembles the schema presented in 6.2. In other words, after the problematic situation is detected, and the top-level plan is halted, the following steps are performed:

- Situation assessment: the recovery module analyzes the problematic situation to determine whether the situation is recoverable according to table 7.1. If the answer is yes, it formulates a belief state that contains the different hypotheses for which of the perceived objects corresponds to the requested one. This step is achieved by considering the properties of the requested object and of the perceived objects to generate. Unlike the situation assessment step in 6.2, here we consider a belief state that consists of a set of possible worlds with probabilities.
- Planning: the planner is called to achieve the goal of disambiguating the situation and anchoring the requested object.
- Plan execution: the plan is executed, and either the requested object is found and identified and can be anchored, or it is established that it cannot be identified.
- Monitoring: if during the execution of the recovery plan, new perceived objects are encountered that completely or partially match the primary or a secondary object, the the whole process is started again.
- Resuming execution: if recovery was successful, the execution of the top-level plan is resumed.

7.3 Situation Assessment

Typically, ambiguous situations in anchoring involve partial matches of some perceived candidate objects. This occurs when a certain property or relation is observed with uncertainty, or neither it nor its opposite was observed at all. Situation assessment involves considering how probable it is that those properties hold. There are two sources from which this information can be obtained.

First, the vision system (and other sensing systems) can provide us with matching degrees, which can serve as weights when assigning a probability distribution to a property or relation. For instance, if it could not be determined whether the color of a perceived object π_1 was red or orange, but red is matching better than orange, we might assert in our planner language `(color pi-1 = (red 0.6) (orange 0.4))`, which is equivalent to $P(color(\pi_1) = \texttt{red}) = 0.6$ and $P(color(\pi_1) = \texttt{orange}) = 0.4$.

Second, we can use background domain-knowledge expressed as conditional probabilities. This background knowledge is encoded as probabilistic assertion rules that can specify conditional and prior probability distributions over the values of uncertain properties of perceived objects.

7.3.1 Probabilistic Assertion Rules

Probabilistic assertion rules are part of the first-order language used by the planners PTLplan and PC-Shop to write planning domains. Assertion rules are used in action templates to describe the results of the application of an action in a belief state. These action results may involve conditional probabilistic effects, and hence the same representation can also be used to encode background probabilistic knowledge.

Example The following rule describes the conditional probability of a perceived object `?o` containing (`has`) a substance (`milk, tea` or `nothing`) given its shape (`cup, bowl`, or something else).

```
(forall (?o) (percept ?o)
    (cond
         ((shape ?o = cup)
          (has ?o = (milk 0.4)(Tea 0.4)(nothing 0.2)))
         ((shape ?o = bowl)
          (has ?o = (milk 0.2)(Tea 0.3)(nothing 0.5)))
         ((true)
          (has ?o = (Tea 0.1)(nothing 0.9)))))
```

This rule can be used when the robot sees an object but does not know what it contains. For instance, percepts that have the shape of a cup are believed to contain either milk, or tea (with 0.4 probability), or nothing (with 0.2 probability). Conditional probabilities are defined using the `cond` form which specifies conditional outcomes. It works like a LISP `cond`, where each clause consists of a test (which may refer to uncertain properties) followed by a consequent formula.

The `forall` assertion formula allows to iterate over elements of a certain type (here `percept`) to execute an assertion formula (here the `cond` form).

Assertion formulas are applied to belief states, and belief states are probability distributions over sets of possible worlds (crisp states). The assertion formula is applied to each possible world in the belief state, resulting in new possible worlds. These new possible worlds and their probabilities constitute the new belief state(s). Note that the belief state explicitly represents a joint probability distribution over the uncertain properties and relations in the situation.

7.3.2 Creating the Initial Belief State

To generate the initial belief state encoding the unexpected situation of failing to anchor an object o due to ambiguity and given present uncertainties, the following steps are performed: the situation assessment process takes a set of relational anchoring candidates as input and performs the following four steps.

1. **Initialization** A number of properties not related to the anchoring candidates are assessed, such as the location of the robot and the topology of the room. The aim of this step is to include knowledge that is necessary to do planning.

2. **Description generation** The descriptions $Desc(\pi_j)$ and $R_k(\pi_j, \pi_k)$ are computed for all the perceived objects π_j and π_k in the relational anchoring candidates. The properties and relations that are considered are those appearing in $Rel_{desc}(o)$ for the object o such that π_j is a candidate object for o. Uncertain properties are estimated, in the manner described in section 7.3.1, by using matching degrees and background knowledge. The result of the first step is a belief state representing a joint probability distribution over all uncertain properties and relations of the perceived objects present in the anchoring candidates.

3. **Classification** The possible worlds of the belief state are partitioned into three different sets for the definite case and two sets for the indefinite case. For the definite case, one partition contains the possible worlds where there is a unique matching candidate. A second partition contains those worlds where there is a conflict due to the presence of more than one matching candidate, and a third set includes worlds where there is no matching candidate. Partitioning relies on the evaluation of three existential formulas derived from $Rel_{desc}(o)$. Those formulas test if there is exactly one, two or more, and no relational anchoring candidate that matches $Rel_{desc}(o)$, respectively. In these formulas, the object names in $Rel_{desc}(o)$ are replaced by existentially quantified variables.

   When the recovery module deals with an indefinite case, the partitioning yields only two sets: the first set contains worlds where there is no matching candidate, while the second set contains worlds where there is at least a matching candidate (remember that in the indefinite case there is no situation of conflict).

At this stage, additional knowledge can be asserted to specify conditions under which it is possible to acquire information about uncertain properties of anchoring candidates. For instance, if `(marked o)` is an uncertain property of object o, knowledge about where the robot can observe a mark on o, e.g., `(mark-visible o = (loc1 1/2)(loc2 1/2))`, is added to the possible worlds where `(marked o)` holds. In this case, it is stated that the mark on o can be observed with the same probability either from location `loc1` or location `loc2`.

Sometimes, one might want to provide more weight to the possible worlds where there is exactly one matching anchoring candidate. After all, if the robot was ordered to go to "the container with milk" it might be reasonable to consider it likely that there is exactly one such object. Hence, at this step one may discount the probability of the possible worlds with none or too many matching candidates using a discount factor α and then renormalize the probability distribution of the possible worlds to sum to one.

4. **Labeling** The formula `(anchor o` π`)` is added to each possible world where percept π is a fully matching candidate, The formula `(anchor o null)` is asserted in the set of non-matching worlds (and conflict worlds for the definite case) to encode that the desired object o cannot be anchored. The percepts π are those returned in the answers to the existential formulas used to partition the possible worlds in step 3.

Example Consider the situation where the anchoring module failed to anchor the object identified by the symbol c_1 where the desired objects is described as "the container with milk, and which is near the fridge" because there are two perceived container objects π_c and π_b near the fridge π_f such that π_c is a green cup and π_b is a blue bowl. Let's also assume that the recovery module uses the probabilistic conditional rule from section 7.3.1.

In step one, it is asserted that the robot is at the entrance of the coffee room, the locations of the different perceived objects, and so on.

Step 2 consists of computing the description of all the perceived objects appearing in the relational candidates and relations among them to build an initial belief state for the subsequent steps.

There are two relational candidates: $(\pi_c\ (\pi_f))$ and $(\pi_b\ (\pi_f))$. The descriptions we obtain are a set Rel of two relations

$$Rel = \{(\text{near}\ \pi_c\ \pi_f), (\text{near}\ \pi_b\ \pi_f)\}$$

and the following descriptions of the different perceived objects:

$$Desc(\pi_c) = \{(\texttt{percept}\ \pi_c), (\texttt{shape}\ \pi_c = \texttt{cup}), (\texttt{color}\ \pi_c = \texttt{green})\}$$
$$Desc(\pi_b) = \{(\texttt{percept}\ \pi_b), (\texttt{shape}\ \pi_b = \texttt{bowl}), (\texttt{color}\ \pi_b = \texttt{red})\}$$
$$Desc(\pi_f) = \{(\texttt{percept}\ \pi_f), (\texttt{shape}\ \pi_f = \texttt{fridge})\}$$

We assert these descriptions, apply the background knowledge rule in section 7.3.1 and obtain a belief state bs with four possible worlds $s_1, \ldots, s_4$ such that:

$$\begin{aligned}
s_1 = \{ \quad & (\texttt{has}\ \pi_c = \texttt{milk}), (\texttt{has}\ \pi_b = (\texttt{tea}\ 3/8)(\texttt{nothing}\ 5/8))\} \\
s_2 = \{ \quad & (\texttt{has}\ \pi_c = \texttt{milk}), (\texttt{has}\ \pi_b = \texttt{milk})\} \\
s_3 = \{ \quad & (\texttt{has}\ \pi_c = (\texttt{tea}\ 4/6)(\texttt{nothing}\ 2/6)), \\
& (\texttt{has}\ \pi_b = (\texttt{tea}\ 3/8)(\texttt{nothing}\ 5/8))\} \\
s_4 = \{ \quad & (\texttt{has}\ \pi_c = (\texttt{tea}\ 4/6)(\texttt{nothing}\ 2/6)), (\texttt{has}\ \pi_b = \texttt{milk})\}
\end{aligned}$$

In addition, they all contain $Rel \cup Desc(\pi_c) \cup Desc(\pi_b) \cup Desc(\pi_f)$.

The probability distribution over the possible worlds is:

$$p(s_1) = 0.4 \cdot 0.8 = 0.32;\ p(s_2) = 0.4 \cdot 0.2 = 0.08$$
$$p(s_3) = 0.6 \cdot 0.8 = 0.48;\ p(s_4) = 0.6 \cdot 0.2 = 0.12$$

The probability $p(s_1)$ of possible world s_1 is computed as the joint probability of π_c containing milk and π_b not containing milk. The same process is applied to compute the probabilities of worlds s_2, s_3, and s_4.

In step 3, the situation assessment process classifies the worlds according to the number of matching candidates. In worlds s_1 and s_4 there is one and only one matching candidate. In world s_2 there are two matching candidates, therefore we have a conflict. Finally in world s_3 there is no candidate object matching $Rel_{desc}(c1)$, i.e., the test formula for no matching candidates

$$\begin{aligned}
\neg\exists x_1, x_2\ (\quad & (\texttt{type}\ x_1 = \texttt{container}) \wedge (\texttt{has}\ x_1 = \texttt{milk}) \wedge \\
& (\texttt{near}\ x_1\ x_2) \wedge (\texttt{shape}\ x_2 = \texttt{fridge}))
\end{aligned}$$

holds in s_3.

In step 4, the situation assessment module adds the formulas $(\texttt{anchor}\ c_1\ \pi_c)$ to s_1, $(\texttt{anchor}\ c_1\ \pi_b)$ to s_4, and $(\texttt{anchor}\ c_1\ \texttt{null})$ to both s_2 and s_3.

7.4 Planning to Gather Information

Once the recovery module has created the belief state encoding the possible worlds, it passes it to the planner together with the goal of finding an anchor to the requested object. The goal formula specifies that the robot must know which percept, or none, to anchor the object symbol to.

The goal is achieved once a specific recovery action `(anchor` *o x*`)` has been performed. It represents the decision to anchor the symbol *o* to some specific perceived object *x* (or to no object at all if *x* = `null`). This action has as a precondition that *x* is the only remaining anchor for *o*, i.e., the truth value of the formula `(nec (anchor` *o x*`))` is true. Thus, all other candidate anchors have to be eliminated before the `anchor` action is applied.

The generated plans incorporate also motion actions and sensing actions aiming at collecting more information about the unknown properties of the perceived objects. The following PTLPLAN action template, for instance, is used to look for marks (and other visual characteristics) on perceived objects.

```
(ptl-action
   :name     (look-at ?y)
   :precond  (((?p) (robot-at = ?p))
              ((?y) (perceived-object ?y)) )
   :results  (cond
               ((and (marked ?y = t)
                     (mark-visible-from ?y = ?p))
                (obs (marked ?y = t)))
               ((not (and (marked ?y = t)
                          (mark-visible-from ?y = ?p)))
                (obs (marked ?y = f)))) )
```

In short, the `precond` part states that the action requires a perceived object `?y` and a current position `?p`. The `result` part states that if `?y` is actually marked, and if the robot looks at `?y` from the position from which the mark is visible, then the robot will observe the mark (and thus know that there is a mark), and otherwise it will not observe any mark. The `obs` form is the way to encode that the robot makes a specific observation, and different observations result in different new belief states. In this case, there would be one belief state where the robot knows there is a mark, and one where it knows there is no mark on the side observable from current position `?p`. If the robot keeps making observations, it can ideally eliminate anchoring hypotheses (signified by `(anchor` *o x*`)`) until only one remains. It can then perform the action `(anchor` *o x*`)`.

The generated recovery plan has a conditional form where each branching follows an information-gathering action,i.e, each branch corresponds to one the planning-time belief states produced by the corresponding information

gathering action. Basically, the condition to branch on is nothing else but the observation associated with the branch belief state. The following plan is generated to recover from an ambiguous situation where the anchoring module could not anchor a perceived green gas bottle because it was not known whether it has a mark. The percept of the gas bottle is named `pi-1`, while the positions used where the robot moves to observe the gas bottle are named `r1-2`, `r1-3`, and `r1-4`.

```
((move r1-2) (look-at pi-1)
    (cond
      ((marked pi-1 = f) (move r1-3) (look-at pi-1)
          (cond ((marked pi-1 = f)
                    (move r1-4)
                    (look-at pi-1)
                    (cond ((marked pi-1 = t)
                              (anchor gb1 pi-1) :success)
                          ((marked pi-1 = f)
                              (anchor gb1 null) :success)))
                ((marked pi-1 = t)
                    (anchor gb1 pi-1) :success))
      ((marked pi-1 = t)
         (anchor gb1 pi-1) :success)))
```

7.5 Execution and Monitoring of Recovery Plans

The execution of the information-gathering plan is carried out by the plan executor described in chapter 3. Basically, non observation actions are translated into low-level sensori-motoric processes that control the motion of the robot. For instance, the action `(move r1-2)` is translated to a set of navigation behaviors including a behavior to reach location `r1-2` and a behavior to avoid obstacles. These behaviors are executed in parallel.

Observation actions are translated into sensing processes. For instance, in our implementation, the execution of the `(look-at pi-1)` action is carried out by a process that points the on-board camera in the direction of `pi-1` to be able to capture images to be used by the on-board vision system. The vision system, extracts percepts and their properties (color, shape, position prop) and sends them to the perception module. The evaluation of branching conditions are carried out by procedures that query the perception module about the truth values of the observation predicates. The `anchor` action has a special role: it causes the symbol of the requested object to be anchored to a specific perceived object.

If the recovery is successful, i.e., it is possible to anchor the requested object to a perceived object, the plan executor resumes the execution of the top-level plan. Otherwise, a permanent failure is raised. As a result, The recovery module

can start another recovery episode whose objective is to search for the requested object in other areas of the work space of the robot.

Closed World Assumption Monitoring

The recovery plans are generated under the assumption that all relevant candidates have been observed. However, there may be additional candidate objects that are not visible from the initial position of the robot, but become visible as the robot moves around while executing its recovery plan. The anchoring system regularly checks for the appearance of new percepts matching the description of the requested object. If such a percept is detected, the assumption of knowing all relevant objects has been violated, and the current recovery plan is considered obsolete, and therefore dropped. Hence, a new initial belief state is produced taking into account the previous perceived objects and the information gained about them as well as the new perceived object properties. A new recovery plan is generated for starting from the new initial belief state and reaching a belief state where the anchoring recovery goal is satisfied.

Replanning for new candidate objects is not only used for the unexpected discovery of a new object, it is also instrumental in the case where the robot was explicitly searching for a candidate, and found a partially matching one. In such a situation, one cannot jump to the conclusion that the correct object has been found, but must plan to find more information about the object in question (together with the other remaining candidate objects).

7.6 Multi-Episode Planning

There are situations where a complete recovery plan is difficult to generate because the number of anchoring candidates and their uncertain properties is very high, thereby increasing the complexity of the planning process. Another reason might be that the planner does not have enough time to generate a complete plan. One way to deal with such situations is to generate plans that disambiguate the unexpected situation with a threshold probability instead of plans that always have to succeed. Both PTLplan and PC-Shop are capable of generating such plans.

An alternative option is to have several planning episodes where in each episode a short plan to solve part of the planning problem is generated and executed. As recovering from ambiguous anchoring situations consists in collecting information, required to eliminate hypotheses about different anchoring candidates, one could design a recovery strategy where the gathering of such information is distributed over different planning/execution episodes. In each episode, a plan is generated and executed to collect only a piece of the total information. For instance, in the example of the containers above, we might have two episodes such that in the first one a short plan is generated and executed to check whether the perceived cup π_c contains milk. If the execution of the plan

reveals that the perceived cup π_c does not contain milk, a second plan can be generated and executed to check if the object to anchor is the bowl π_b, i.e., by checking its content.

Therefore, a multi-episode process of recovering from anchoring ambiguous situations could be performed in the following steps:

1. A belief state bs encoding the ambiguous situation is created as outlined above (section 7.2).
2. The recovery module determines what piece of missing information I is to be collected taking into account the current ambiguous situation.
3. The planner is called to generate a plan to collect I starting from bs.
4. The plan is then executed resulting in a new situation. As a result, bs is updated to reflect the new situation.
5. The recovery module checks whether the new situation is still ambiguous. If the answer is yes, a new information-gathering episode is started from step 2. Otherwise, the recovery procedure terminates by either determining the candidate object to anchor or asserting that there is no such object.

Selecting the Information to Collect

We have already seen that the process of recovering from anchoring failures is a hypothesis elimination process. Each recovery action is meant to acquire some information that helps in reducing the uncertainty in the belief state encoding the anchoring ambiguous situation. One measure that can be used to determine what information should be gathered, so uncertainty in the belief state can be reduced, is the information theoretic measure of information gain discussed in section 6.6.

For the purpose of recovering from anchoring ambiguous situations, information gain can be used to select the information to collect so that the uncertainty of anchoring the requested object is minimized. This can be done by taking into account the uncertainty about the different properties of the different anchoring candidates. The uncertain property that achieves the highest information gain is then selected as the information I to collect. In other words I is computed as follows:

$$I = \underset{F_i}{\operatorname{argmax}}\, IG(A|F_i)$$

where A is a binary random variable such that $A = 0$ means that the requested object o cannot be anchored, whereas $A = 1$ means that the requested object o

can be anchored. F_i are random variables associated with the different uncertain properties such that the values of each F_i are those of the corresponding property.

The probability mass functions of the different random variables (i.e., A and F_i) as well as the conditional probabilities of A given F_i are computed from the belief state bs that encodes the ambiguous situation. In particular, we have

$$P(A = 0) = \sum_{s \in bs; s \models (\texttt{anchor o null})} p_{bs}(s)$$

and $P(A = 1) = 1 - P(A = 0)$.

Example To clarify our ideas, let's consider the ambiguous situation given in the example of section 7.3.2 above.

To determine what uncertain property the robot should gather information about, we compute $IG(A|F_c)$ and $IG(A|F_b)$ where F_c, respectively, F_b, is a random variable whose values denote the possible contents of the perceived cup π_c, respectively bowl π_b. The probability mass function of the random variable A is computed as follows:

$$\begin{array}{rcl} P(A=0) & = & p(s_2) + p(s_3) \\ & = & 0.56 \\ & \text{and} & \\ P(A=1) & = & p(s_1) + p(s_4) \\ & = & 0.44 \end{array}$$

This result is computed based on the fact that the requested object o can be anchored in possible worlds s_1 and s_4, but not in possible worlds s_2 and s_3.

The entropy of A is computed as:

$$\begin{array}{rcl} H(A) & = & -(P(A=0) \cdot log(P(A=0)) + P(A=1) \cdot log(P(A=1))) \\ & = & 0.3 \end{array}$$

$P(A|F_c)$ is computed from the initial belief state and is given in the following conditional probability table:

	$F_c = \text{milk}$	$F_c = \text{tea}$	$F_c = \text{nothing}$
$P(A=0\|F_c)$	$\frac{0.08}{0.4} = 0.2$	$\frac{0.32}{0.4} = 0.8$	$\frac{0.16}{0.2} = 0.8$
$P(A=1\|F_c)$	$\frac{0.32}{0.4} = 0.8$	$\frac{0.08}{0.4} = 0.2$	$\frac{0.04}{0.2} = 0.2$

Consequently, we get

$$IG(A|F_c) = 0.3 - 0.22 = 0.08$$

Using the same procedure for computing $IG(A|F_b)$ we get

$$IG(A|F_f) = 0.3 - 0.29 = 0.01$$

Since $IG(A|F_c)$ is greater than $IG(A|F_b)$, the information to collect is the content of the perceived cup π_c. Therefore, in the first episode, the planner is called to find a plan to eliminate hypotheses about the content of π_c. Note that a second episode is needed to check the contents of the bowl π_b, no matter what the result of executing the plan of the first episode. This is because we are dealing with a definite case. If we are dealing with an indefinite case, a second episode is needed only when the execution of the plan of the first one reveals that π_c does not contain milk.

7.7 Test Scenarios

In this section, we describe test scenarios that we ran to show the feasibility of using sensor based planning as a tool to recover from ambiguous situations in anchoring. The test scenarios were run using Pippi, our Magellan Pro robot, in a lab indoor environment. It is worth mentioning that the time taken to generate recovery plans was always less than two seconds.

7.7.1 Anchoring under Uncertainty

The aim of this test scenario is to demonstrate the ability of the approach to handle cases were there is no recovery plan that is certain to succeed. Here, the plan executor is trying to execute the top-level action `(go-near gb1)` where the symbol `gb1` refers to a marked gas bottle (indefinite reference). From her initial position, Pippi could perceive two gas bottles π_1 and π_2, but no mark is perceived on them (figure 7.1). The mark could be on one of four sides of each gas bottle. However, the presence of obstacles prevented observing them from all sides.

This situation resulted in a failure to anchor the requested object, and thus recovery was needed. The situation assessment step produced a belief state consisting of four possible worlds $s_1, \cdots, s_4$ with a uniform probability, i.e., $p(s_i) = 1/4; 1 \leq i \leq 4$. Each world reflected if π_1 or π_2 was having a mark, and if it had a mark, on which side it was. A plan was generated in which the robot was to move to the different accessible positions to look for a mark on the perceived gas bottles (right picture of figure 7.1). When observation actions were performed, the probabilities of the possible worlds were updated accordingly. For instance, if no mark was seen on one side of π_1, the probability that

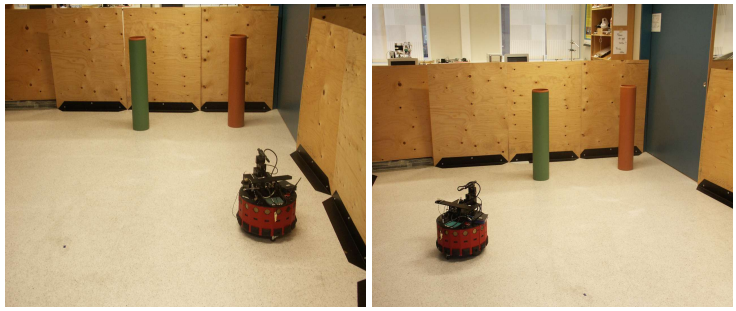


Figure 7.1: Trying to identify a marked gas bottle. Left: From her initial position, Pippi perceives two candidate objects resulting an anchoring ambiguous situation. Right: Pippi trying to recover from the ambiguous situation by moving to a new observation position to look for a mark on both candidates.

π_1 is a match decreased. In one of the runs, with the configuration shown in figure 7.1, Pippi had not found a mark after observing π_1 from three sides and π_2 from two sides. At this stage, the anchoring module decided to anchor the more likely candidate π_2 but with a low degree of certainty. The scenario was also run successfully under different configurations in terms of the locations of the gas bottles, whether they were marked or not, and on which sides of the gas bottles the marks were placed.

7.7.2 Handling of Newly Perceived Candidates

This scenarios was run to test how run-time newly perceived objects were handled. Again, Pippi was to approach a gas bottle with a mark upon it (indefinite reference). This time, only one gas bottle π_1 was initially perceived but without a mark on it (see the left image in figure 7.2).

As a first step of recovering from the resulting situation, the situation assessment produced a belief state with two uniformly distributed possible worlds s_1 (π_1 is marked), and s_2 (π_1 is not marked). In this scenario, all the other three sides of π_1 were considered to be observable, thus in s_1 there was a probability distribution for which side, of the gas bottle, had the mark. The generated recovery plan included moving to three predetermined observation locations (`r1-2`,`r1-3`, and `r1-4`) to observe the mark on π_1. The plan is the same as the one given in section 7.4 above.

Next, Pippi started the execution of the recovery plan and moved to the first observation position `r1-2`, so she could look for a mark on π_1. Pippi could not see any mark on π_i, yet she detected a second gas bottle π_2 that was initially occluded by the first one (see the right image in figure 7.2). This situation was a manifestation of a violation to the closed world assumption that π_1 was the only anchoring candidate. Therefore, the execution of the recovery plan was

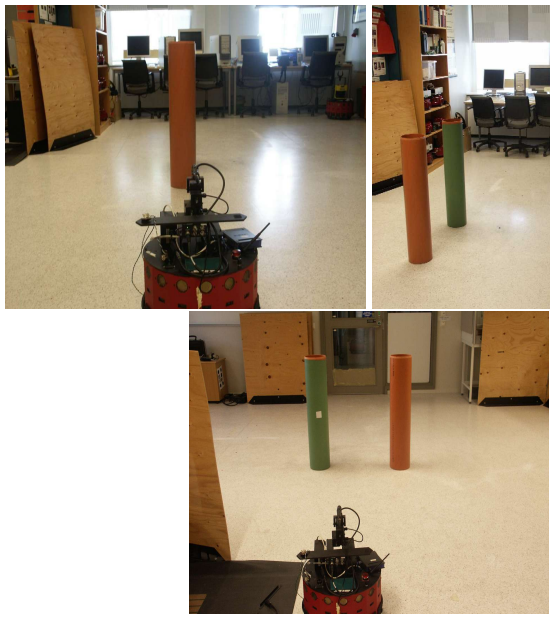


Figure 7.2: Trying to identify a marked gas bottle. Left: From her initial position, Pippi perceives only one candidate object π_1 but without a mark. Right: Pippi perceives a second candidate π_2, while trying to observe a mark on the first one from another observation position. Bottom image: Pippi observes a mark on the second candidate π_2, which results in anchoring the requested object to π_2

halted and a new recovery episode was started to take into account the newly perceived candidate π_2. The situation was reassessed and a new recovery plan was produced that included actions to look for marks on both the old and the new gas bottle. The bottom image of figure 7.2 shows a situation where Pippi could see the mark on the second candidate π_2, thereby resulting in anchoring the requested object to π_2.

This scenario was also successfully run under different configurations in terms of which of the bottles was marked, and which side of the bottle the mark was on.

7.7.3 Including Background Knowledge

In this scenario, background knowledge was used to resolve an ambiguous situation where planning time was very short. Pippi had to achieve the same task as before, but in addition there was the background knowledge that 90% of brown gas bottles have a mark on them, and only 20% of green ones have a mark. From its initial position, Pippi perceived two gas bottles: one green π_1 and one brown π_2. Using the background knowledge, the situation assessment procedure resulted in a belief state where π_2 had a higher chance of matching the description and therefore it was decided to be anchored directly.

7.8 Summary and Conclusions

Detecting and responding to execution failures is a necessary step for building robust-execution systems of robot plans. This chapter presented a case study of dealing with one important class of failures that may arise while executing a symbolic plan by mobile robot. Our focus was on studying failures that are due to ambiguous situations in anchoring perceptual information to symbols, i.e., situations where there are several hypotheses about which perceived object can be anchored. We showed that such situations are characterized by uncertainty due to lack of information about properties of perceived candidate objects.

The chapter presented an active probabilistic recovery procedure to handle such failures. The process of recovery relies on an important step of situation assessment to build a probabilistic multi-hypothesis representation of the occurring failure situation. Recovery is, then, achieved by computing an information-gathering plan whose successful execution permits to draw conclusions about which percept(s) the object of interest can be anchored to. This recovery procedure is a direct application of the information-gathering schema described in chapter 6 to collect information needed for the semantic knowledge-based execution monitoring process. Here, however, the information to collect is needed by the plan executor itself in order to correctly execute a symbolic action.

The use of probabilities was shown to make it possible to distinguish between more likely and less likely candidate percepts as well as handling cases where the ambiguous situation could only partially be resolved. This means

that the anchoring module can choose to anchor an object to a symbol with a degree of certainty; absolute certainty is not required. The original anchoring framework by Coradeschi and Saffiotti [37] did not permit that. In fact there are situations where certain relevant properties of an object cannot be determined, and absolute certainty about which is the correct object is not attainable. Besides, the robot might have only a limited amount of time for planning.

As it has been already discussed in chapter 6, using sensor-based planning might result in an issue of scalability to handle situations that involve a high number of hypotheses about the actual state of the world. We presented an alternative solution to tackle such issue by adapting a recovery strategy that plans to gather only small pieces of information at a time. Certainly, this strategy might result in sub-optimal recovery solutions. Nevertheless, it has the advantage of solving recovery problems that are unsolvable if all the required information has to be planned for at once.

Another potential issue with our encoding of the uncertainty about ambiguous situations is the non-compact representation of uncertainty in belief states. In fact, this is not a shortcoming of the approach itself, as other tools can be used to encode belief states. These include tools that use Bayesian networks to represent first order knowledge [58, 79] and the work by Milch *et al.* [101] that uses Bayesian Networks to explicitly represent objects and relations among them.

Chapter 8
Experiments

In this chapter, we describe the experimental set-up used to validate the proposed approaches of monitoring the execution of plans using semantic domain-knowledge. Our experiments were conducted both in simulation as well as in real world using a real robot. The aim of conducting simulation experiments was to collect large amounts of data for the purpose of statistically evaluating the performance of both monitoring processes: crisp *SKEMon* (presented in chapter 4) and probabilistic *SKEMon* (covered in chapter 5). The evaluation is based on viewing each *SKEMon* process as a classification system that predicts the outcome of the execution of an action, i.e., success or failure. Thus, performance is measured using information about actual and predicted classifications done by the *SKEMon* process. On the other hand, the real-world experiments were aimed at demonstrating the practicability of the proposed approaches onboard real robots.

Before describing the simulation experiments, we start by presenting some of the real-robot test scenarios so that the reader can be well acquainted with the whole process. Note that test scenarios for detecting and responding to anchoring failures are presented in chapter 7.

8.1 Real-Robot Test Scenarios

The real-robot test scenarios were performed using Pippi, the Magellan Pro mobile robot, as the main protagonist. Both *SKEMon* processes (i.e., crisp and probabilistic) as well as the information gathering process described in chapter 6 were implemented as part of the hierarchical plan executor described in 3.3.

Our test scenarios consisted in performing navigation tasks in a house environment. The house comprises 4 rooms, named `r1` to `r4`, which are asserted in the semantic knowledge base as: `r1` and `r2` are of type bedroom, `r3` is of type living-room, and `r4` is of type kitchen (see figure 8.1). In each room, there are furniture items that are specific to the type of that room. Objects that are not specific to any type of room are also present, e.g., tables, plants, etc. The se-

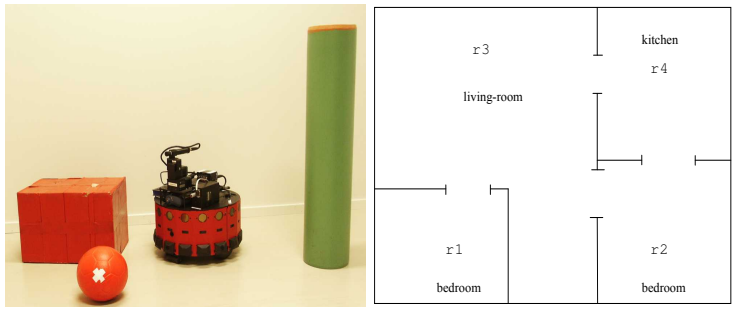


Figure 8.1: Experimental setup. (Left) Pippi, the Magellan Pro robot together with simple objects used to represent furniture items. (Right) Map of the environment used in performing test cases.

mantic knowledge base contains the same concept definitions as the ones used in simulation experiments (see appendix A.1). To take into account partial observability of the environment, we set up LOOM to use open-world semantics.

Since our main objective is to show the capacity of execution monitoring using semantic knowledge and not object recognition, we let objects with simple shapes like cylinders and boxes stand in for beds, sofas, etc. The experiments reported below have been performed in a lab environment, placing the simple objects above to simulate pieces of furniture. It should be noted that our approaches do not depend on the simple vision system we used. One can use more robust object recognition and classification systems such as the system described in [127] that uses scale and orientation invariant local descriptors (SIFT features) [92] to identify objects occurring in typical household environments.

Perceptual information is produced by an on-board vision system that gets images from a frame grabber connected to the on-board CCD color camera. Each image is segmented using mean shift techniques [34] in the LUV color space. Objects are detected by a process that tries to match the resulting segments to a set of contours of simple objects (box, ball, cylinder, garbage can, cup,...). The vision system also computes properties of the matched objects such as color, relative position, and enclosed regions (marks). The computed perceptual information is then sent through TCP/IP to the perception and anchoring module where data about perceived objects is managed, i.e., stored and maintained through tracking. The perception process checks whether a perceived object is a new object to add it together with its perceived properties to the local database of perceived objects. As a result, all requests issued by the *SKEMon* process regarding perceptual information are handled through retrieving information from the database of perceived objects.

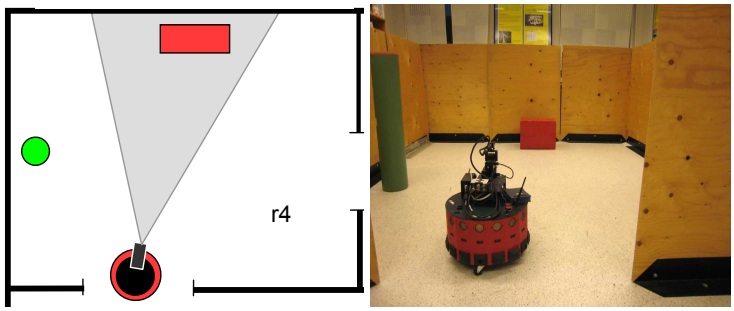


Figure 8.2: A scenario where Pippi has just finished executing the action `(enter r4)` to enter the kitchen. (Left) Two objects are placed in room `r4`: one oven (red box) and one sink (green cylinder). From its current place, Pippi sees only the oven. (Right) Picture of Pippi seeing the oven (red box).

8.1.1 Crisp *SKEMon* Test Cases

We start by describing test cases where the crisp version of the semantic knowledge based execution monitoring is used to check the implicit expectations of navigation actions. The tasks assigned to the robot were to clean the different rooms in the house. The plans used to achieve those tasks were all generated under the assumption that actions are deterministic and that there is no noise in sensing data. The task plans included two types of navigation actions `(goto ?d)` and `(enter ?loc)`. The former were used to put the robot in a position in front of a door `?d` that leads to a room `?loc`, while the latter were used to cause the robot to move inside a room `?loc`. As doors did not have any semantic knowledge associated with them, we will only describe the monitoring process of the `(enter ?loc)` actions. The *SKEMon* process used a credulous approach whenever it could not determine if the room where the robot ended up was of the same class as the asserted one.

Correct Success Result 1

In this test case, Pippi was in the living room (`r3`) and was asked to clean room `r4` (the kitchen). The generated task plan specified that she should enter `r4` through door `d4`. Once Pippi finished the execution of `(enter r4)`, the *SKEMon* process was called to check if the current room was a kitchen. The available perceptual information indicated that Pippi saw only an oven (represented by a red box in figure 8.2). After asserting this fact, LOOM classified the current room as a kitchen. The classification was supported by the constraint that ovens are to be found only in kitchens. As a result, the *SKEMon* process returned "*success*", which was interpreted as an indication of the successful execution of the action `(enter r4)`.

This test case shows that not all the implicit expectations need to be verified in order to conclude that an action has been successfully executed. It suffices to see objects that are exclusively characteristic of a specific class of rooms.

Correct Success Result 2

The aim of this test case is to show that the monitoring process uses the absence of counter evidence to correctly conclude that an action is executed successfully although some implicit expectations are not known to hold or not.

Here, Pippi started in `r4` and executed the action `(enter r3)` to move to room `r3`; she could perceive one sofa (represented by a green box) from her final position. The *SKEMon* process told LOOM about the acquired perceptual information and then queried if the current location was a living-room. The answers of LOOM to the queries showed that the current room was neither a living-room nor a non living-room. Hence, the *SKEMon* process reached the "unknown" monitoring result. As the *SKEMon* process was using a credulous approach and no counter evidence of being in a living-room was detected, the current room was assumed to be a living-room; therefore, it was conclude that the action had been successfully executed. Notice that although the robot saw a sofa, LOOM could not classify the current room as a living-room because sofas were not exclusively constrained to be in living-rooms. This is opposed to the case of ovens,i.e., they were to be found only in kitchens.

False Success Result

The aim of this test case is to show that the credulous approach might cause the monitoring process to wrongly conclude that the execution of an action has succeeded. Here, Pippi was put in the living-room and was told to go to bedroom `r1`. Pippi was placed in an initial position that made the actual execution of the task plan end up in the kitchen instead. After finishing the execution of the action `(enter r1)`, Pippi could see only one table. As tables can be in any room, LOOM could not deduce if the current room was a bedroom. This led the *SKEMon* process to reach the "*unknown*" result. Because there was no counter evidence against being in a bedroom, it was conclude that the action had been successfully executed, i.e., Pippi was assumed to be in `r1`. This was a false positive since in reality Pippi was in the kitchen (room `r4`).

Correct Failure Result

The aim of this test case is to show that it is enough that one implicit expectation is violated to conclude that the execution of the action has failed.

We basically repeated the previous test case, but instead of placing the object that stands for a table in the field of view of Pippi, we put a green cylinder to represent a sink. This meant that after the execution of the `(enter r1)` action,

the *SKEMon* process could conclude that the execution has failed. This result was caused by the violation of the implicit expectation that "bedrooms do not contain objects of type sink".

8.1.2 Probabilistic *SKEMon* Test Cases

In these test cases uncertainty in action effects and sensing was reasoned about by the probabilistic *SKEMon* process. Here, the general and simplified sensing models were employed by the monitoring process. The parameters used by the models were the same as the ones used in the simulation experiments (see section 8.2.4). In all of the test cases, we considered the monitoring of the navigation action `(move ?loc1 ?loc2)` with a model of a prior giving the robot 20% chance of being stuck in room `?loc1` (outcome 1, denoted $R = 1$) and 80% chance of ending up in room `?loc2` (outcome 2, denoted $R = 2$). The *SKEMon* process returned the outcome that had the highest posterior probability as the outcome produced by the execution of the action.

Negative Evidence Against One Outcome

In this test case, we show how acquiring negative evidence changes the prior probability of the outcomes. Here, Pippi started in room `r3` and executed the action `(move r3 r4)` to move from room `r3` to room `r4`. Pippi effectively moved into `r4` and could perceive only one sink from its final place.

The monitoring process was then called to compute the posterior of the outcomes. Both sensing models of the probabilistic monitor gave the same result, i.e., $P(R = 2) = 1.0$ and $P(R = 1) = 0.0$. This result is supported by the fact that room `r3` is a living-room and according to the semantic knowledge base it should contain no sink. Moreover, when using the general sensing model, the only object that could be mistaken as a sink was an oven, which was defined not to be in a living-room either. Therefore, seeing a sink was negative evidence against the first outcome.

Negative Evidence Against All Action Outcomes

The aim of this test scenario is to show that the observations acquired by the robot might reveal an exception that it is outside of the action model. Here, Pippi started in room `r3` and executed the action `(move r3 r1)`, but instead of ending up in either `r1` or `r3` as specified by the model, she ended up in the kitchen (room `r4`).

As in the previous case, the only object perceived by Pippi was a sink. Computing the posterior, using both sensing models, resulted in $P(R = 1) = 0$ and $P(R = 2) = 0$. This result was supported by the fact that room `r1` was of type bedroom and room `r3` was of type living-room, and according to the semantic knowledge base both should contain no sink. Therefore seeing a sink

was negative evidence against both outcomes. This is an exceptional situation, which indicates that something outside of the action model has occurred. In fact, before Pippi started the execution of the action, we changed her location and direction so that instead of entering `r1`, she entered `r4`.

To recover from this exceptional situation, our fall-back strategy was to assume that the robot was in one of the remaining locations, i.e., either room `r2` or room `r4` with equal probability $1/2$ and recompute the posterior as before. This yielded $P(\texttt{(robot-in = r2)}) = 0$ and $P(\texttt{(robot-in = r4)}) = 1.0$, i.e., the robot was certainly in the kitchen.

Uncertain Posterior

We considered three test cases where the acquired perceptual information did not eliminate the uncertainty about the outcomes of the executed action. All the three test cases involved monitoring the execution of the same action, i.e., `(move r3 r1)` to move from the living-room `r3` to bedroom `r1`. In all the test cases, Pippi started from the same location in room `r3` and effectively moved into room `r1`. The only difference between the different runs was the acquired perceptual information.

In the first run, Pippi did not see any object from its final place. Both sensing models of the probabilistic *SKEMon* process gave the same posterior: $P(R = 1) = 0.13$ and $P(R = 2) = 0.87$. As the monitoring module was considering the outcome with the highest probability as the actual outcome, Pippi was concluded to be in room `r1`.

In the second test case, Pippi could see two chairs from its final place. As there was no constraint about the type of the room where chairs could be located, the posterior was once more the same as in the first test case, i.e., $P(R = 1) = 0.13$ and $P(R = 2) = 0.87$. This was a predictable result as the conditional probabilities of seeing chairs in both rooms were the same.

In the third case, Pippi could see only a sofa. The posterior computed using the simplified sensing model was $P(R = 1) = 0.51$ and $P(R = 2) = 0.49$. This meant that the monitoring process returned `r3` as the final location of the robot, which was not the case. On the other hand, the posterior computed using the general sensing model was $P(R = 1) = 0.4$ and $P(R = 2) = 0.6$; therefore, the monitoring process correctly estimated the final location of the robot, i.e., room `r1`. Notice that the difference in the computed posteriors was due to the fact that, when using the general sensing model, seeing a sofa was interpreted as either really seeing a sofa or mistakenly seeing a bed as a sofa; seeing a sofa was not counter evidence against any of the outcomes.

8.1.3 Information Gathering for Crisp *SKEMon*

In this section, we describe test scenarios that show the capacity of the *SKEMon* process to reason about situations involving lack of information. The goal was

to test the sensor-based planning approach, proposed in chapter 6, to compute solutions for gathering information useful for checking implicit expectations. Remember that information gathering is a cautious approach that can be used to handle situations where crisp *SKEMon* cannot establish whether some implicit expectations hold or not, i.e., their truth values are "unknown".

Active Information Gathering

The aim of this test scenario was to show how planning could be used to collect information to infer the truth values of implicit expectations. The assigned top-level task was to clean the living-room (`r3` in figure 8.1), starting from the kitchen `r4`. In this test scenario, we considered a slightly simple definition of the concept of living-room:

```
(defconcept living-room :is
            (and room
                 (:at-least 1 has-tv)
                 (:at-least 1 has-sofa )
                 (:exactly  0 has-sink))
```

The top-level task plan `((enter r3);(clean r3))` was produced to accomplish the assigned task. Next, Pippi executed the action `(enter r3)` to enter room `r3` and could see a TV set inside the room where she ended up. Then, the *SKEMon* process was called to check the implicit expectations of being in a living-room, which resulted in an "unknown" outcome. The reason was because the truth values of the expectations of having at least one sofa and no sink were not known.

Consequently, an active information-gathering process was started to look for sofas and sinks inside the current room. In this test scenario, PC-SHOP was used to generate the information-gathering plan. The plan included actions to move and scan the room from two predetermined places `r3-1` and `r3-2` (see figure 8.3), and is given as follows:

```
((move r3-1)(eval-exactly 0 has-sink r3)
 (cond
    ((exactly 0 has-sink r3 = t)
        (move r3-2)(eval-exactly 0 has-sink r3)
        (cond
           ((exactly 0 has-sink r3 = t)
               (eval-at-least 1 has-sofa r3)
               (cond
                  ((at-least 1 has-sofa r3 = f) :fail)
                  ((at-least 1 has-sofa r3 = t) :success)))
           ((exactly 0 has-sink r3 = f) :fail)))
```

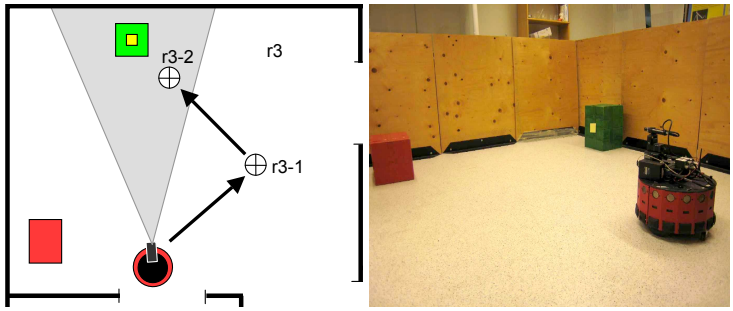


Figure 8.3: A scenario where Pippi has just entered the living-room `r3`. (Left) Pippi sees only a TV set (marked green box), which results in an "unknown" monitoring result. An information-gathering plan is generated to scan the room from two places, `r3-1` and `r3-2`, to look for sofas and sinks. (Right) Picture of Pippi scanning the room from place `r3-1` where she could see a sofa (the red box on her left side).

```
((exactly 0 has-sink r3 = f) :fail)))
```

We had two test cases of this scenario. In the first case, Pippi executed the information-gathering plan and reached a `:success` action. That meant that all the implicit expectations were verified. Therefore, the room where Pippi was located was correctly found to be a living-room, thus the task-plan action `(enter r3)` was concluded to have been executed successfully and the execution of the top-level task plan was resumed with the next action, i.e., `(clean r3)`.

In the second case, we modified the room by adding an object of type sink and removing the object representing the sofa. As a result, the execution of the information-gathering plan failed reached the predicted `:fail` action, because a sink was perceived inside the room. That meant that the implicit expectation of having no sink in a living-room was found to be violated. Consequently, the monitoring process concluded that the task-plan action `(enter r3)` had failed to execute successfully.

Recursive Information Gathering

In this test scenario, we show how the framework applies recursively to monitor the execution of an information gathering plan (itself generated to monitor expectations of a task-plan). We also show how the approach applies to actions other than navigation. In this test scenario, a bedroom was defined as

```
(defconcept bedroom :is
            (:and room
                  (:at-least 1 has-bed)
                  (:at-most  1 has-sofa)))
```

The definition of the relation `has-sofa` was also modified as follows:

```
(defrelation has-sofa :domain room
                      :range  two-seater-sofa)
```

where the new concept `two-seater-sofa` refers to sofas that have exactly two seats, i.e.,

```
(defconcept two-seater-sofa :is
            (and sofa (= number-of-seats 2)))
```

Here, Pippi executed a navigation task-plan that included an action to enter the bedroom `(enter r1)`. From her final place, Pippi did not see any sofas nor any beds. This situation resulted in generating and executing an information-gathering plan to look for sofas and beds inside `r1`. The information-gathering plan is similar to the one in section 6.4.4. Using semantic knowledge to monitor the execution of `(eval-at-most 1 has-sofa r1)` observation action involved deriving the implicit expectation that any perceived sofa inside `r1` had to verify the constraint `(= number-of-seats 2)`. Consequently, a new information-gathering plan was generated, every time a new sofa with an unknown number of seats was perceived. The goal of the new information-gathering plan was to check whether the perceived sofa had a number of seats equal to two by moving in front of the sofa and observing it.

This test scenario shows that the approach leads to a form of interleaving of planning and execution. In fact, in the task-plans, the planner did not include any actions to check the number of seats of sofas. This was handled at run-time by the monitoring process whenever it was needed, i.e., once a sofa with an unknow number of seats was perceived.

8.1.4 Information Gathering for Probabilistic *SKEMon*

We also ran test scenarios to handle situations where the computed posterior of the action outcomes by probabilistic *SKEMon* involved information gathering as discussed in sections 5.4 and 6.6. A typical scenario where information gathering was needed was while executing the following conditional task-plan by Pippi to clean the living-room (room `r3`) starting from room `r1` (bedroom).

```
((move r1 r3)
 (cond
     ((robot-in = r3) (clean r3) :success )
     ((robot-in = r1) (move r1 r3)
                      (cond
                          ((robot-in = r3) (clean r3) :success)
                          ((robot-in = r1) :fail )  ))))
```

As with the other test scenarios of probabilistic *SKEMon*, the same model of the movement action was used, i.e., the execution of `(move r1 r3)` could result in two alternative outcomes. In the first outcome ($R = 1$), the robot stays unintentionally in room `r1` with probability 0.2, while in the second outcome, ($R = 2$), the robot moves effectively into room `r3` with probability 0.8.

As the plan executor needed a crisp answer about the location of the robot to continue the execution of the plan, the *SKEMon* process returned the outcome that had a posterior probability greater than a threshold $T = 0.8$. If no outcome satisfied the criterion, an information-gathering process was launched to gather information that was likely to reduce the uncertainty in the posterior of the outcomes. If, after the information gathering, there was still no outcome that satisfied the selection criterion, the outcome with the highest probability was returned to the plan executor.

When Pippi executed the first action `(move r1 r3)`, she could see only one table from her final place. Consequently, the computed posterior of the action outcomes using the simplified sensing model was:

$$P(R = 1) = 0.29; P(R = 2) = 0.71$$

As none of the outcomes had a posterior probability greater than 0.8, an information-gathering episode was started to look for objects in order to reduce the uncertainty about the location of the robot. To this end, information gain was computed for the outcomes of the action given the different values of observation variables. This resulted in selecting objects of type sofa to look for, as on average observing sofas was predicted to achieve the highest information gain. Therefore, Pippi moved to a location in the middle of the room and scanned it looking for sofas.

We ran two test cases of this scenario, with both starting in the same way (i.e., by executing the `(move r1 r3)` action and observing a table), but continued differently in the information-gathering phase. In the information-gathering phase of the first test case, Pippi could see a sofa, and hence the posterior probabilities of the two outcomes were as follows:

$$P(R = 1) = 0.06; \quad P(R = 2) = 0.94$$

As $P(R = 2) \geq T$, the probabilistic *SKEMon* process returned `r3` to the plan executor as the resulting outcome of executing action `(move r1 r3)`. That meant that the next action of the task-plan to execute was `(clean r3)`.

In the information-gathering phase of the the second test case, Pippi did not see any object except the table. Thus, the posterior that was computed before the information gathering did not change, i.e., $P(R = 1) = 0.29$ and $P(R = 2) = 0.71$. Because neither outcome had a posterior greater than 0.8, the *SKEMon* process returned the one that had the highest posterior. In other words, `r3` was returned to the plan executor as the resulting outcome of executing action `(move r1 r3)`.

8.2 Simulation Results

Our simulation experiments consisted of two scenarios taking place inside a simulated house environment: a manipulation scenario and a navigation scenario. A simulated mobile robot called Astrid, of type ActiveMedia PeopelBot, was used as the main protagonist in both scenarios (see figure 8.5). The robot was equipped with a simulated pan-tilt color camera that was used to acquire visual perceptual information about the environment. The simulation experiments were run using the 3D robot software simulator "Gazebo" [84].

8.2.1 Performance Evaluation Metrics

Due to lack of benchmarks in the area of execution monitoring of symbolic plans, we base our evaluation on the metrics of false positive rate (FPR) and true positive rate (TPR). Both metrics assume a binary classifier that tries to classify a set of instances as either *positive* or *negative*. FPR is defined as the ratio between the number of negative instances that are erroneously classified as positive (FP) and the total number of actual negative instances (N), that is,

$$FPR = \frac{FP}{N}$$

On the other hand, TPR is defined as the ratio between the number of positive instances that are correctly classified as positive (TP) and the total number of actual positive instances (P), i.e.,

$$TPR = \frac{TP}{P}$$

One way to evaluate the performance of classifiers is to analyze their results using Receiver Operating Characteristic (ROC) graphs [46]. A ROC graph is a plot of the rate of false positives (FPR) versus the rate of true positives (TPR), such that the X axis represents FPR and the Y axis represents TPR. A perfect classifier will achieve an FPR that is equal to zero and a TPR that is equal to one, i.e., point $(0, 1)$ in ROC space. A totally random classifier will have an FPR that is equal to TPR, i.e., a point on the line $y = x$, which is called the line of no-discrimination. A classifier that has its (FPR, TPR) point under the line of no-discrimination is considered to be a bad classifier, while a classifier that has its (FPR, TPR) above that line is considered to be a good classifier. Figure 8.4 shows an example of a ROC graph where the (FPR, TPR) points of three classifiers are plotted.

Other metrics related to FPR and TPR include *accuracy* and *precision*. The metric of accuracy gives the proportion of all the instances that are correctly classified, i.e.,

$$accuracy = \frac{TP + TN}{P + N}$$

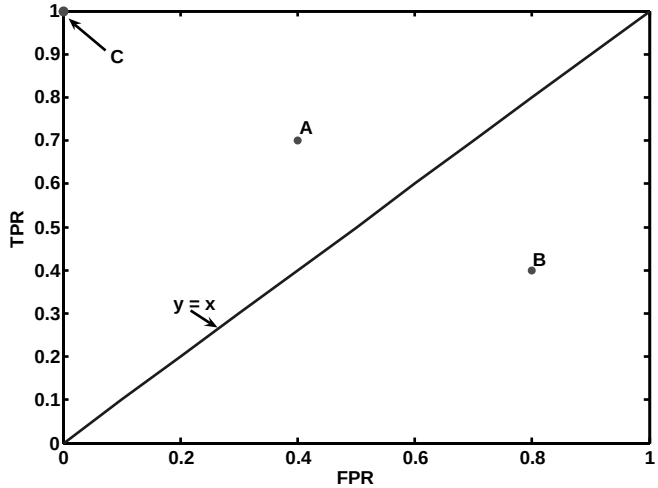


Figure 8.4: An example of a ROC graph showing three classifiers A, B, and C. A is a good classifier, since its (FPR, TPR) point is above the line of no-discrimination, i.e., $y = x$. B is a bad classifier, as its (FPR, TPR) point is under the line of no-discrimination. C is a perfect classifier, since it correctly classifies all positive instances and does not mistakenly classify any negative instance as positive.

where TN represents the number of negative instances that are correctly classified as negative. The metric of precision gives the proportion of the correctly classified positive instances:

$$precision = \frac{TP}{TP + FP}$$

8.2.2 Manipulation Scenario

In the manipulation scenario, we used a simulation of the smart house described by Saffiotti and Broxvall in [132]. Besides the robot, the experimental set-up included a two prismatic-joint arm that was attached to the roof of a fridge (see figure 8.5). The arm was used to achieve manipulation tasks aiming at picking up objects inside the fridge so that they could be placed on the base of the robot (e.g., to carry them to another location inside the house). Inside the fridge, there could be objects of different types. We considered objects that were instances of either of cup, glass, bottle, bowl, or box types. All of these types were defined as `containers` that had some specific properties, which were expressed in terms of constraints over relations to other objects of atomic concepts (handle, cap, and cover). The semantic definitions of such types are relatively simple, and they are given in appendix A.1.

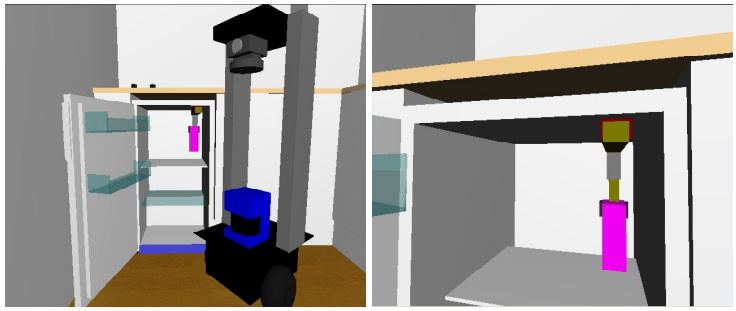


Figure 8.5: Simulation experimental setup. (Left) Astrid, the robot, near the fridge where an arm is picking up an object. (Right) A close-up of the arm while picking up an object inside the fridge.

We used the two simple solids of a cylinder and a box to represent a container, while the related objects (i.e., handle, cover, cap) were represented by marks of different colors placed on the container. For example, an object of type cup was represented by a box (container) that has a yellow mark (handle).

8.2.3 Navigation Scenario

In this scenario, Astrid was acting in a house environment that comprised rooms of different types (bedroom, living-room, kitchen, bathroom, office, and utility room). In each room, there were furniture items, which were typical of the type of that room. For instance, in a kitchen, there's at least one oven, at least one sink, etc. and in an office there's at least one PC, at least one chair, etc. In total, there were thirteen different types of objects that could exist in a room. The semantic definitions of the different types of rooms and furniture items is given in appendix A.1.

The semantic knowledge used in this scenario is more complex than the one used in the manipulation scenario. The definitions contain more constraints, and there are more related objects to take into account in order to classify a room. Moreover, there are situations where seeing objects does not contribute to the classification process. For instance, seeing only an object of type plant does not help the robot identify in which room it is, as plants can be in any room. As in the manipulation scenario, the furniture items were represented by objects of simple shapes and colors.

8.2.4 Parameters Used in Probabilistic *SKEMon*

Parameters of the Manipulation Domain

The state variables of the manipulation domain represent the number of handles (S_1), covers (S_2), and caps (S_3) that can be related to a container. As the knowledge base shows, the maximum number of objects that a container can have is always one. This means that all these variables have the same domain, i.e., the set $\{0, 1\}$. The probability values of the state functions $p(s_j|r)$ of the different state variables S_j are either 0 or 1 depending on the class of the object to pick up. For example, if r models the outcome where the arm picks up a cup, then $p(s_1 = 0|r) = 0$ and $p(s_1 = 1|r) = 1$ while $p(s_j = 0|r) = 1$ and $p(s_j = 1|r) = 0$ for the other two state variables S_j.

Regarding the sensing models, we used ad-hoc values of the different parameters of the binomial and multinomial probability mass functions. The probability parameters of the binomials used in the simplified sensing model were all fixed to 0.8. The probability parameters of the multinomial functions, i.e., $p(\mathbf{g}_i|s_i)$ used in the general sensing model to encode how objects are classified when seen are given as follows:

$C_1 =$ handle	$p_1 = 0.8$	$p_2 = 0.0$	$p_3 = 0.0$	$p_4 = 0.2$
$C_2 =$ cover	$p_1 = 0.0$	$p_2 = 0.6$	$p_3 = 0.2$	$p_4 = 0.2$
$C_3 =$ cap	$p_1 = 0.0$	$p_2 = 0.2$	$p_3 = 0.6$	$p_4 = 0.2$

Recall that for a certain object of class C_j, p_i expresses the probability of (mis)classifying that object under class C_i. In this case, the probability of correctly classifying a handle is $p_1 = 0.8$, whereas the probability of correctly classifying a cover is $p_2 = 0.6$. Similarly, the probability of misclassifying a cover as a cap is given as $p_3 = 0.2$, etc. The probability of missing (not seeing) an object is p_4 for all three classes.

Parameters of the Navigation Domain

There are thirteen state variables, such that each variable represents the number of objects of a certain atomic class, e.g., bed. The domains of those variables range from zero to a certain maximum number; they are given as follows:

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
Object	bed	sofa	sink	oven	table	tv	chair
max #	2	2	2	1	2	1	4

	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}
Object	tub	fridge	plant	PC	clothes-dryer	washing-machine
max #	1	1	3	1	1	1

As described in section 5.3, the values of the state functions $p(s_j|r)$ were set in accordance with the available semantic domain-knowledge, which is given in appendix A.1. The values of the state functions were either 0 or 1 with the exception of the following functions:

	$i=0$	$i=1$	$i=2$
$P(S_1=i\|\text{bedroom})$	0.0	0.7	0.3
$P(S_2=i\|\text{living-room})$	0.0	0.6	0.4
$P(S_3=i\|\text{kitchen})$	0.0	0.4	0.6
$P(S_5=i\|\text{kitchen})$	0.0	0.8	0.2

These values were subjective and reflect one's belief of having a certain number of objects of a specific type in the different types of rooms.

We also assigned the probability 0.8 to all the parameters of the binomials of the simplified sensing model. For the general sensing model, only few types could be misclassified; their classification probabilities are given as follows:

bed	$p_1=0.8; p_2=0.1; p_{14}=0.1$
sofa	$p_1=0.2; p_2=0.7; p_{14}=0.1$
sink	$p_3=0.7; p_4=0.1; p_{14}=0.2$
oven	$p_3=0.1; p_4=0.7; p_{14}=0.2$
table	$p_5=0.8; p_6=0.1; p_{14}=0.1$

where p_i expresses the probability of classifying a seen object as an instance of class C_i while p_{14} represents the probability of missing (not seeing) an object in a room where the robot is located. The sensing of all the other objects followed the simplified sensing model, i.e., they were either seen or not seen but not mistakenly seen as objects that were instances of other classes.

8.2.5 Perceiving the Environment

Often, when the robot senses the environment it only gets partial information about the presence of objects and their properties, e.g., due to occlusions. To take this into account, ee model partial observability of the environment using a parameter P_{perc} that specifies the probability of perceiving all objects related to the actual outcome of an executed action. In our experiments, we assume that the process of observing one object is independent of observing another. We also assume that all the objects have the same probability of being perceived, i.e., $\sqrt[m]{P_{perc}}$, where m is the total number of objects that are related to the actual outcome of the action.

8.2.6 Crisp *SKEMon* Results

We tested the performance of the crisp *SKEMon* process on both scenarios. For the manipulation scenario, each experiment consisted of executing the high-level action "`(pick-up obj)`" by the arm to pick up the object, identified by the symbol "obj", inside the fridge. The high-level definition of the pick-up action is given as follows:

```
(ptl-action
  :name    (pick-up ?obj)
  :precond (((?obj)(object ?obj)
                   (and (arm-empty = t)(inside-fridge ?obj))))
  :results (and (arm-empty = f)(holding = ?obj)) )
```

The *SKEMon* process was called once the low-level execution process reported that it succeeded in picking up the desired object. The task of the *SKEMon* process consisted in verifying that the object actually picked up by the arm was of the same type as the desired one. To do so, the camera on-board the robot was used as the main sensing modality to acquire perceptual information necessary for monitoring. That meant that the robot had to be in front of the fridge facing one side of the picked up object. Consequently, the robot could observe the related object (handle, cap, or cover) if it was on the side facing the robot.

The "`(pick-up obj)`" experiment was run such that `obj` was asserted 100 times as an instance of each of the five classes cup, bowl, bottle, glass, or box. That gave a total of 500 runs where in each run the type of the object that the arm ended up picking up was uniformly sampled from the five available types. The perception by the robot of the related object (e.g., handle, cap,...) to the picked up one was decided by generating a random number from a Bernoulli distribution with probability P_{perc}. In other words, the related object was on the side facing the robot if the sampled random number was less than P_{perc}.

Similarly, we conducted a total of 600 runs of the `(enter loc)` navigation action to enter a room identified by the symbol `loc` and whose type was asserted to be one of the available room types, i.e., bedroom, living-room, etc. Each room type was considered 100 times; each time, the type of the actual final location of the robot was sampled uniformly from the six available types. A world state, containing objects that were consistent with the actual location, was then generated using the state functions used in the probabilistic version of *SKEMon* (see section 8.2.4). Which objects could be perceived from the robot's place were determined according to the parameter P_{perc} . The perceivable objects were then put in places where the robot could see them while the others were hidden. The object detection operation was tuned so that all perceivable objects were actually detected and correctly classified.

Table 8.1 shows the obtained results for three different values of P_{perc}: 0.3, 0.5, and 0.7. The rows of the table represent the ground truth, such that the

Table 8.1: Results of crisp *SKEMon* in monitoring the execution of the actions `pick-up` and `enter`. The rows represent the ground truth, and the columns represent the result returned by the *SKEMon* process, i.e., *success* (S), *failure* (F), and *unknown* (U).

		$P_{perc} = 0.3$			$P_{perc} = 0.5$			$P_{perc} = 0.7$		
		S	F	U	S	F	U	S	F	U
Navigation	S	20	0	70	22	0	71	27	0	58
	F	0	412	98	0	418	89	0	437	78
Manipulation	S	9	0	101	22	0	87	22	0	81
	F	0	75	315	0	128	263	0	163	234

Table 8.2: The rates of true positives (TPR) and false positives (FPR), given as percentages, of crisp *SKEMon* for the actions `pick-up` and `enter`. Two approaches are considered: non-credulous (N-C) treating *unknown* as a separate third case and credulous (C) treating *unknown* as *success*.

		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
	Approach	FPR	TPR	FPR	TPR	FPR	TPR
Navigation	N-C	0	22.22	0	23.65	0	31.67
	C	19.21	100	17.55	100	15.14	100
Manipulation	N-C	0	8.18	0	20.18	0	21.35
	C	80.76	100	67.26	100	58.94	100

first row for each scenario represents the positive cases, i.e., the runs where the expected outcome of the action is the same as the actual outcome. The second row represents negative cases, i.e., runs where the expected outcome of the action is different from the actual outcome. The columns, on the other hand, represent the results of the *SKEMon* process, i.e., **S**uccess, **F**ailure or **U**nknown.

The results show that crisp *SKEMon* is able to get a zero percent of false positives and a zero percent of false negatives. This is a predictable result as perceptual information is assumed to be perfectly reliable although incomplete. We can also remark that the monitor is able to detect most of the failure situations (true negatives) for the navigation action (80, 82, and 85 %). However, for the pick-up action, smaller percentages are detected (19, 33, and 41%).

The high percentages achieved in the navigation scenario are explained by the fact that most concept definitions are highly constrained, and therefore the perception of objects as counter evidence is more likely. As a result, the number of cases where the monitor declares *unknown* is reduced. On the other hand, the definitions of concepts in the manipulation domain involve only a small number of constrains over related objects. Consequently, the probability of not observing counter evidence is higher, which results in having more situations where the type of the picked up object is not known. Moreover, as the constraints involved in the definitions of the concepts bowl and glass are the same,

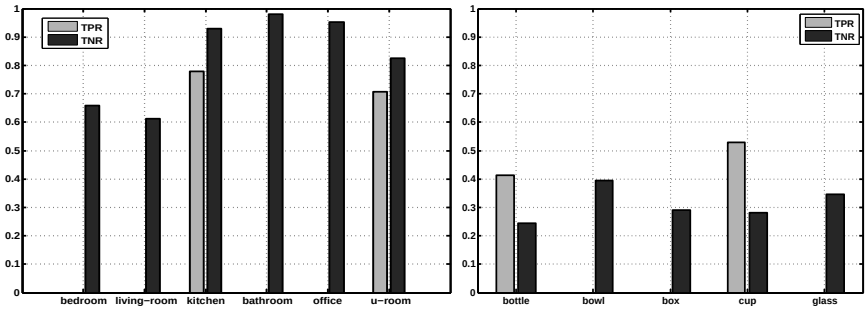


Figure 8.6: Rates of true positives (TPR) and true negatives (TNR) achieved by crisp *SKEMon* for different types of rooms (left) and containers (right).

all situations where the expected outcome is picking up a bowl, but the actually picked up object is a glass (and vice versa) are declared as *unknown*.

The results show also that the percentages of correct detection of *success* are low: around 22, 24, 32% for the navigation action and 8, 20, and 21% for the manipulation action. Notice that all the other cases where the monitor declares *unknown* when the actual execution is successful are due to the fact that we set LOOM to use open-world semantics to classify instances. In other words, `:at-most` and `:exactly` constraints can only be deduced to be violated but not to hold even if the asserted knowledge should make them hold. The reason is that LOOM does not know whether there is more information that could make those constraints not hold. Following the same reasoning, LOOM can only prove that an `:at-least` constraint holds.

Figure 8.6 shows the detailed rates of true positives (TPR) and true negatives (TNR) for the different types of the expected object. One can observe that all detected success cases for the navigation actions are from runs where the robot successfully moved into either the kitchen or the utility-room. On the other hand, the detected success cases for the manipulation actions are from runs where the expected object to pick up was a cup or a bottle. This is due to the fact that the robot could see objects that were defined to be exclusively related to those types of rooms and containers. For instance, seeing an oven was sufficient to conclude that the current room was a kitchen, while seeing a handle on the picked up container caused that container to be classified as a cup (see the definitions of relations in appendix A.1). On the other hand, we can observe that the percentage of correctly detected failures is never zero. This brings us to the conclusion that if the SKB contains constraints that uniquely identify classes of objects, crisp *SKEMon* would be able to detect more successful execution cases, and thus a lower number of *unknown* cases.

When the monitoring process takes a credulous approach, all *unknown* results are counted as *success*. Table 8.2 shows the rates of true positives (TPR) and false positives (FPR) of crisp *SKEMon* treating *unknown* as a separate case,

i.e., using a non-credulous approach, and crisp *SKEMon* using a credulous approach. One can notice that the credulous approach detects 100% of successful execution cases, but at the same time it reports higher rates of false positives especially when applied in the manipulation domain. It should also be noted that all the results of crisp *SKEMon*, both credulous and non-credulous, indicate a good performance (even for low values of P_{perc}). The reason is that when the FPR vs TPR points are plotted in the ROC space, they are all above the line of no-discrimination (i.e., the line representing the function $f(x) = x$). This result is due to the fact that crisp *SKEMon*, with a non-credulous approach, achieves 100% specificity (i.e., it is equivalent to 0% of false positives) and a sensitivity (equivalent to TPR) that is greater than zero. On the other hand, using a credulous approach gives 100% of true positives and a FPR that is less than TPR. Therefore, one can conclude that crisp *SKEMon* is good at detecting execution failures, provided that the defined concepts be sufficiently constrained, but it is less good in detecting correct execution.

8.2.7 Probabilistic *SKEMon* Results

We used the same experimental set-up to run experiments to evaluate the performance of the probabilistic *SKEMon* process. However, this time, we considered probabilistic action models with two possible outcomes. For the pick-up action, the first outcome expressed picking up the desired object, while the second outcome expressed picking up another object that was near the desired one. For the navigation scenario, we considered the action `(move loc1 loc2)` to move the robot from its initial room `loc1` to destination room `loc2`. The first outcome of the action reflected the situation where the robot would remain unintentionally in `loc1`, while the second outcome expressed the case where the robot would effectively end up in room `loc2`.

To simulate the unreliable effects of executing `(pick-up obj)`, we used the prior probability of the two outcomes, specified in the action model, to sample the object that the arm would actually pick up. Then, the sampled object was placed in a location where the arm would pick it up. Similarly, the unreliable effects of executing `(move loc1 loc2)` were simulated by sampling the room, where the robot actually ended up, from the two rooms `loc1` and `loc2` using the prior probability specified in the action model. The process of determining which objects were perceivable by the robot was done the same way as in the crisp *SKEMon* experiments. To simulate unreliable sensing, the acquired perceptual information was corrupted according to the parameters of the sensing model used by the monitoring process.

Probabilistic *SKEMon* was evaluated using four values of P_{perc}: 0.1, 0.3, 0.5, and 0.7. For each value of P_{perc}, three prior probability distributions of the action outcomes were considered: $\{p(r_1) = 0.8, p(r_2) = 0.2\}$, $\{p(r_1) = 0.5, p(r_2) = 0.5\}$, and $\{p(r_1) = 0.2, p(r_2) = 0.8\}$. We performed experiments where the two objects, or rooms, involved by the two action outcomes could

Table 8.3: Results of probabilistic *SKEMon* using the simplified sensing model to monitor the execution of navigation and manipulation actions with two possible outcomes. The rows represent the actual outcome of the executed action while the columns represent the outcome predicted by the monitoring process.

		$P_{perc} = 0.1$		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
		r_1	r_2	r_1	r_2	r_1	r_2	r_1	r_2
Navigation	r_1	2470	241	2394	216	2466	215	2528	208
	r_2	254	2435	251	2539	206	2513	208	2456
Manipulation	r_1	1204	671	1277	626	1327	505	1466	406
	r_2	696	1179	571	1276	544	1374	427	1451

Table 8.4: Results of probabilistic *SKEMon* using the general sensing model to monitor the execution of navigation and manipulation actions with two possible outcomes.

		$P_{perc} = 0.1$		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
		r_1	r_2	r_1	r_2	r_1	r_2	r_1	r_2
Navigation	r_1	2467	257	2470	227	2456	201	2514	227
	r_2	239	2437	225	2478	234	2509	196	2463
Manipulation	r_1	1135	688	1269	605	1379	518	1474	428
	r_2	726	1201	603	1273	522	1331	462	1386

be asserted to be of any of the available types. For each combination of types, we repeated the experiment 50 times. This resulted in a total of 3750 runs for the manipulation action and 5400 runs for the navigation action.

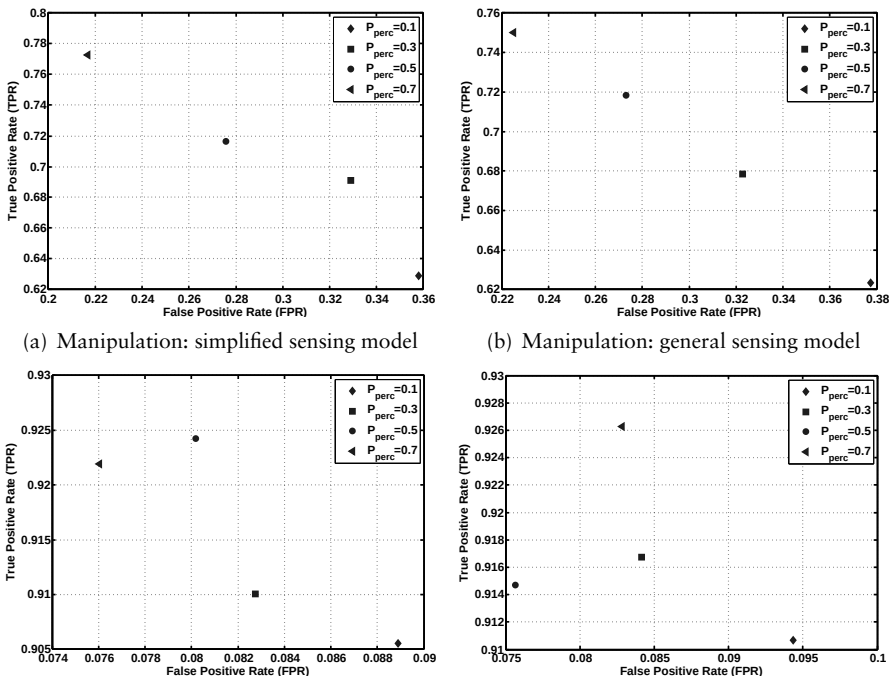
The results of probabilistic *SKEMon* using the simplified sensing model and the general sensing model are shown respectively in tables 8.3 and 8.4. In both tables, the rows represent the ground truth, and the columns show the results of the *SKEMon* process, which were computed by selecting the outcome with the higher posterior probability (ties were broken randomly). The rates of true positives (TPR) and false positives (FPR) for both scenarios and for both sensing models are given in table 8.5. Both rates are computed by considering outcome r_2 as the positive case and outcome r_1 as the negative case. The ROC graphs representing the (FPR, TPR) points are shown in figure 8.7.

The results with both sensing models indicate good performance as TPR tends to be high, while FPR tends to be low. The results show also that the performance of probabilistic *SKEMon* using the simplified sensing model is slightly better than the one with the general sensing model. This is due to the fact that when the simplified sensing model is used, simulated objects can be either seen or missed but not misclassified. On the other hand, when the general sensing model is used, simulated objects can be missed as well as misclassified due to the introduction of noise in perception.

As in the case of crisp *SKEMon*, we notice that the performance of probabilistic *SKEMon* is very good in the navigation scenario. The percentages of

Table 8.5: The rates of true positives (TPR) and false positives (FPR) of probabilistic *SKEMon* for the two types of actions using the two sensing models (S: simplified; G: General). The rates are given as percentages.

		$P_{perc} = 0.1$		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
	Model	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR
Navigation	S	8.89	90.55	8.28	91.00	8.02	92.42	7.60	92.19
	G	9.43	91.07	8.42	91.68	7.56	91.47	8.28	92.63
Manipulation	S	35.79	62.88	32.90	69.08	27.57	71.64	21.69	77.26
	G	37.74	62.32	32.28	67.86	27.31	71.83	22.50	75.00



(a) Manipulation: simplified sensing model

(b) Manipulation: general sensing model

(c) Navigation: simplified sensing model

(d) Navigation: general sensing model

Figure 8.7: The upper left corner of the ROC graphs representing the FPR vs TPR points of probabilistic *SKEMon*. The points are shown for the different values of P_{perc}.

Table 8.6: Accuracy of probabilistic *SKEMon* in monitoring the execution of navigation and manipulation actions using the two sensing models (S: simplified; G: General). The results are given as percentages.

	Model	$P_{perc} = 0.1$	$P_{perc} = 0.3$	$P_{perc} = 0.5$	$P_{perc} = 0.7$
Navigation	S	90.83	91.35	92.20	92.30
	G	90.81	91.63	91.94	92.17
Manipulation	S	63.55	68.08	72.03	77.79
	G	62.29	67.79	72.27	76.27

Table 8.7: Precision of probabilistic *SKEMon* in monitoring the execution of navigation and manipulation actions using the two sensing models (S: simplified; G: General). The results are given as percentages.

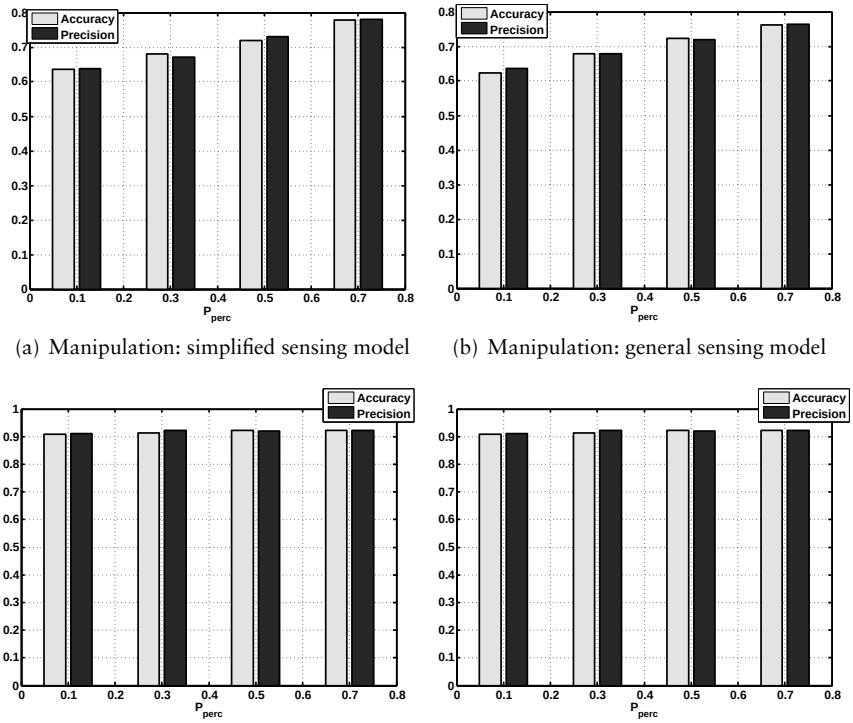
	Model	$P_{perc} = 0.1$	$P_{perc} = 0.3$	$P_{perc} = 0.5$	$P_{perc} = 0.7$
Navigation	S	90.99	92.16	92.12	92.19
	G	90.46	91.61	92.58	91.56
Manipulation	S	63.73	67.09	73.12	78.14
	G	63.58	67.78	71.98	76.41

false positives are as low as 7.6%, while the percentages of true positives are as high as 92.63%. The performance in the manipulation scenario is not as good as in the navigation scenario, nevertheless it gets better when the probability P_{perc} is higher. As explained above, this is due to how definitions of concepts are constrained as well as how much of the environment is observable, i.e., the value of P_{perc}. Compared to crisp *SKEMon*, probabilistic*SKEMon* achieves better results thanks to its ability to reason about uncertainty in both sensing and action effects.

The results of accuracy and precision are given in tables 8.6 and 8.7. One can observe that probabilistic *SKEMon* is highly accurate in detecting successful and failed execution in the navigation scenario, but it is less accurate in the manipulation scenario. As with FPR and TPR, accuracy and precision get better when the environment is more observable (P_{perc} is higher). Figure 8.8 shows accuracy and precision results, given as bar plots, of probabilistic *SKEMon*.

Corrupted Action Model

To test the influence of incorrect actions models on the performance of probabilistic *SKEMon*, we ran experiments where the action model used by probabilistic *SKEMon* was corrupted. In other words, the probabilities assigned to the possible outcomes were not always correct. In these experiments, we considered only the monitoring of the execution of navigation actions using the general sensing model. The action model used by *SKEMon* had a prior giving



(a) Manipulation: simplified sensing model (b) Manipulation: general sensing model

(c) Navigation: simplified sensing model (d) Navigation: general sensing model

Figure 8.8: Bar plots showing accuracy and precision of probabilistic *SKEMon* in monitoring the execution of navigation and manipulation actions. The results are shown for different values of P_{perc}.

Table 8.8: Results of performance of probabilistic *SKEMon* using the general sensing model in monitoring the execution of navigation actions with perfect and corrupted action models.

		$P_{perc} = 0.1$		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
		r_1	r_2	r_1	r_2	r_1	r_2	r_1	r_2
Perfect Model	r_1	496	217	549	210	564	199	543	197
	r_2	66	2821	35	2806	39	2798	21	2839
Corrupted Model	r_1	869	382	895	346	894	304	949	312
	r_2	49	2300	32	2327	23	2379	16	2323

Table 8.9: Comparison of the rates of true positives (TPR) and false positives (FPR) of probabilistic *SKEMon* using perfect and corrupted navigation-action models.

	$P_{perc} = 0.1$		$P_{perc} = 0.3$		$P_{perc} = 0.5$		$P_{perc} = 0.7$	
	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR
Perfect Model	30.43	97.71	27.67	98.77	26.08	98.63	26.62	99.27
Corrupted Model	30.54	97.91	27.88	98.64	25.38	99.04	24.74	99.32

a probability of 0.2 to the first outcome and a probability of 0.8 to the second outcome, i.e., $\{p(r_1) = 0.2, p(r_2) = 0.8\}$.

The experiments were performed such that the two rooms involved by the two action outcomes could be asserted to be any of the available room types. For each combination of types, the experiment was run 100 times, and each time the final room of the robot was sampled from the two rooms `loc1` and `loc2`. In one half of the 100 runs, the sampling was performed according to the prior probability of the outcomes specified in the action model. In the other half, the sampling was from a uniform probability distribution, i.e., both modeled outcomes had the same probability of being the actual outcome.

To be able to analyze the results of probabilistic *SKEMon* with a corrupted action model, we repeated the same experiments where the action model was not corrupted. Table 8.8 summarizes the results achieved by the *SKEMon* process when both action models are used (i.e., corrupted vs non-corrupted models). The corresponding rates of false positives (FPR) and true positives (TPR) are given in table 8.9 and plotted in a ROC graph in figure 8.9.

The results indicate that the performance, in terms of FPR vs TPR, of probabilistic *SKEMon* with a corrupted action model is comparable to that when a perfect action model is used. However, as demonstrated by tables 8.10 and 8.11, *SKEMon* with a corrupted action model is both less accurate and less precise than *SKEMon* with a perfect action model. Nevertheless, the difference in accuracy and precision is minimal (see figures 8.10 and 8.11). This leads us to conclude that semantic knowledge helps in building reliable execution monitoring systems even when the action model is corrupted.

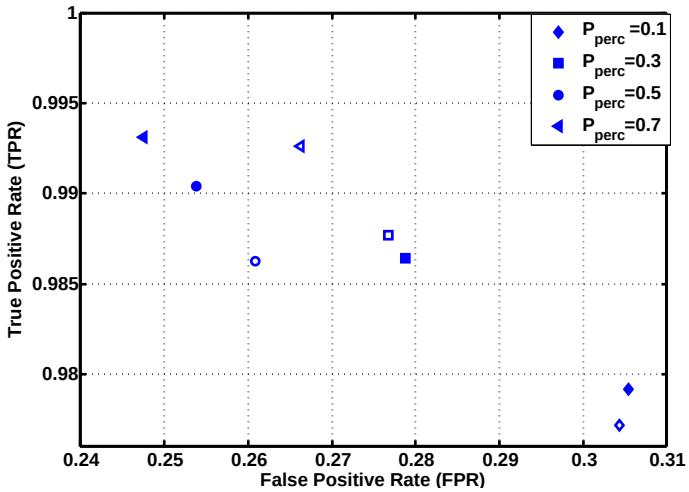


Figure 8.9: Upper-left corner of the ROC diagram showing the FPR vs TPR points of probabilistic *SKEMon* using corrupted and perfect action models. The points are for monitoring the execution of navigation actions. Filled figures represent points when the corrupted model is used while empty figures represent points when the perfect model is used.

Table 8.10: Comparison of the accuracy of probabilistic *SKEMon* using corrupted and perfect navigation-action models. The results are given as percentages.

	$P_{perc} = 0.1$	$P_{perc} = 0.3$	$P_{perc} = 0.5$	$P_{perc} = 0.7$
Perfect Model	92.14	93.19	93.39	93.94
Corrupted Model	88.03	89.50	90.92	90.89

Table 8.11: Comparison of precision results of probabilistic *SKEMon* using corrupted and perfect navigation-action models. The results are given as percentages.

	$P_{perc} = 0.1$	$P_{perc} = 0.3$	$P_{perc} = 0.5$	$P_{perc} = 0.7$
Perfect Model	92.86	93.04	93.36	93.51
Corrupted Model	85.76	87.06	88.67	88.16

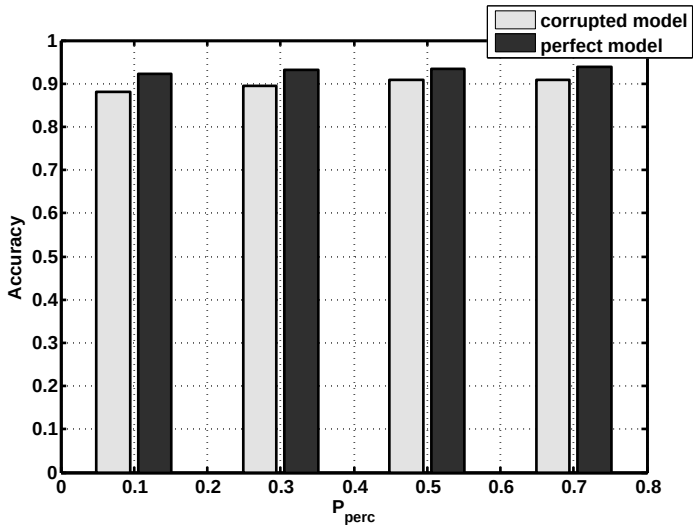


Figure 8.10: Accuracy of probabilistic *SKEMon* using corrupted and perfect action models. The results are for monitoring the execution of navigation actions.

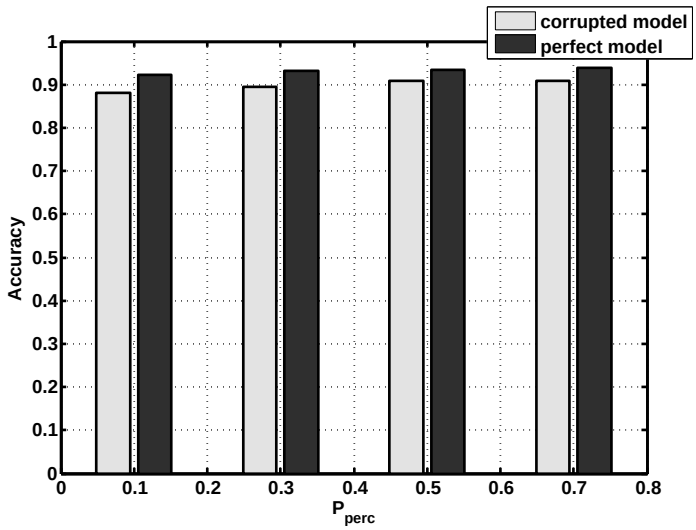


Figure 8.11: Precision of probabilistic *SKEMon* using corrupted and perfect action models. The results are for monitoring the execution of navigation actions.

8.3 Discussion

The main focus of this chapter was to demonstrate that semantic domain-knowledge helps robots achieve good performance in monitoring the execution of their symbolic plans. To this end, we implemented the two monitoring processes (crisp and probabilistic *SKEMon*) on a real robot, and we ran test scenarios involving the execution of symbolic plans generated to navigate in an indoor environment. We also presented and analyzed simulation and experimental data using known statistical performance metrics, i.e., rates of true positives (TPR) and false positives (FPR) as well as accuracy and precision.

Although we have evaluated the performance of our approaches in monitoring the execution of simple actions using a simple vision system, we expect them, especially the probabilistic one, to perform very well in more realistic scenarios when more powerful perception systems are used. This expectation is based on the fact that the probabilistic approach uses a sensing model that is able to reason about noisy sensing where objects can be missed or misclassified when they are detected.

The results reported in this chapter show that semantic domain-knowledge can effectively help robots achieve good performance in monitoring the execution of symbolic plans. In particular, we have seen that crisp *SKEMon* is very good at detecting execution failures if the used semantic knowledge contains enough counter-evidence constraints in the definitions of classes of objects. This result can be used as the basis for selecting constraints to check using active information gathering (see chapter 6). In other words, whenever crisp *SKEMon* returns the "unknown" outcome regarding the execution of a plan action (because of lack of information), only counter-evidence constraints can be selected for generating plans to check them. The objective is to simplify the planning problem when there is a high number of implicit expectations that involve collecting information.

Moreover, the results show that crisp *SKEMon* never declares that implicit expectations are violated when in reality they are not. This claim is supported by the obtained results showing that crisp *SKEMon* achieves zero percent of false negatives independently of how implicit expectations with unknown truth values are treated (i.e., using a credulous approach or not). Certainly, this result is due to the simplifying assumption in our experiments that perceptual information is reliable, i.e., detected objects are not misclassified. Although, we did not notice any performance issues regarding the size of the knowledge bases that we used in our experiments, we expect that with the use of the state-of-the-art knowledge representation systems our approach would be able to handle much bigger and more complex knowledge bases. Our claim is supported by the developments of DL-based systems that are shown to scale up well [67, 103].

When the monitor considers uncertainty in sensing and action effects, the performance is even better than the one achieved by crisp *SKEMon*. This claim

is supported by the high rates of true positives and the low rates of false positives achieved in monitoring the execution of two different types of actions. The reason for the better performance can be attributed to the fact that expectations are no longer treated in a boolean manner, i.e., satisfied, violated, or unknown. Furthermore, probabilistic *SKEMon* bases its decisions about whether the execution of an action has failed or succeeded taking into account how likely each expectation is verified or violated given the acquired perceptual information. Additionally, the results show that even when the action model is corrupted probabilistic *SKEMon* is still able to perform very well. This claim is supported by the data reported in table 8.8.

On the down side, crisp *SKEMon* is unable to detect failure situations when the expected object has similar conceptual description as the execution-time object. A typical example of such situations in our experimental set up is when the robot is expecting to pick up a glass but ends up grasping a bowl. Such situation will always result in "*unknown*" monitoring result. This has also the implication of erroneously considering such cases as resulting in "*success*", when crisp *SKEMon* takes a credulous approach. One way to cope with such an issue is to define concepts to be totally exclusive by providing more constraints involving the properties of their instance objects.

Similar situations arise when both the expected object and the execution-time object are instances of the same class. For instance, the robot wants to move to a bedroom `r1`, but it ends up in another bedroom `r2`. This is typically a situation that cannot be handled by considering only general semantic knowledge. One would need to provide extra information about the expected object (e.g., the size of `r1`), or objects related to it (`r1` has a red bed) to be able to remedy such unexpected situations.

It should be noted that the performance of probabilistic *SKEMon* also drops when applied in such situations, but it does to a lesser extent. Basically, the resulting posterior of action outcomes is highly influenced by the prior probabilities given in the action model. In the case where the expected and the execution-time objects are instances of the same class, the posterior probabilities of the outcomes will always be the same as the prior ones.

Chapter 9
Discussions and Conclusions

This thesis has addressed the topic of *robust* execution of symbolic plans by autonomous mobile robots acting in indoor environments. Our primary focus has been on the ability of a mobile robot to monitor the execution of its plans to detect unexpected situations at execution-time. The thesis has also addressed the issue of responding to such unexpected situations with a special emphasis on situations of lack of information that is required for correct execution of robot actions.

We have claimed that autonomy requires that mobile robots be able to monitor the execution of their actions to make sure that they are executed successfully. In particular, we have identified that existing approaches of execution monitoring of symbolic plans relied mainly on the results of checking the explicit effects of plan actions, i.e., effects encoded in the action model. We have argued that this relies on the hidden assumption that the effects to monitor are directly observable. Our claim has been that such an assumption is not always practical in real-world environments where checking expectations is a complex process. Therefore, we have proposed to increase the reliability of the process of execution monitoring by incorporating more advanced forms of reasoning. In particular, we proposed to use semantic domain-knowledge as a source of information to derive implicit expectations about the effects of actions, and to monitor these expectations using the available perceptual information. This has allowed us to introduce the new notion of Semantic Knowledge-based Execution Monitoring *SKEMon* in mobile robotics. To the best of our knowledge, our work is the first one to use semantic knowledge for the purpose of plan execution monitoring. In fact, with the exception of few cases, e.g., [44, 60, 73, 116, 142], there is little work of using semantic knowledge in mobile robotics in general.

We have proposed a general algorithm for *SKEMon* based on the use of description logics for representing and reasoning about domain knowledge. The choice of using description logics was motivated by their ability to express general knowledge about classes of objects through a concise representation. In

addition, description logics permit the derivation of implicit knowledge from the explicitly represented one; thus, a lot of information about domain entities can be kept implicit. Moreover, description logics are fairly expressive yet supported by efficient inference mechanisms, making them practically useful.

Uncertainty is a feature that is ever present in robotics, and therefore we have addressed the issues of probabilistic uncertainty in action effects, sensing, and world states in the context of using semantic knowledge to monitor plan execution. Being able to reason about uncertainty, an execution monitor can deduce whether a specific action outcome is more likely given the acquired perceptual information, which is generally noisy and incomplete. Therefore, we have considered uncertainty in sensing through a model that expresses the probability of what is observed for a given state of the world. The sensing model permits to state whether an object that exists in the real world is seen or not, by taking, e.g., occlusion into account. The model also accounts for misclassification of objects when they are seen.

Our experimental results have shown that semantic knowledge can contribute to effective plan execution monitoring techniques especially when uncertainty in action effects and sensing is explicitly taken into account. It is worth mentioning that although several of the examples and experiments presented in this thesis involved the robot's location, the problem addressed by our work should not be confused with self-localization. The aim of our work is to monitor the execution of actions by observing their (explicit and implicit) effects: these effects may include the robot's location in the case of navigation actions, but they include other aspects of the world state for other actions, like observation and grasping actions.

In this thesis, we have also argued that autonomy requires that robots be able to respond to unexpected situations on their own. Our argument is motivated by the fact that a robot that can figure out on its own how to cope with unexpected situations is a robot that can continue functioning toward achieving successfully its assigned tasks. In this thesis, we considered unexpected situations caused by lack of information that is necessary for the correct execution of plans. In chapter 6, we presented an information-gathering schema to deal with situations that are characterized by lack of information relevant to semantic knowledge-based execution monitoring. Such information is needed to evaluate implicit expectations whose truth values could not be known using the immediately available perceptual information. We have claimed that gathering the missing information is a cautious approach that can be used when the robot does not want to take a credulous approach, i.e., consider the execution of an action successful as long as no counter evidence is detected.

The proposed information-gathering schema includes steps for modeling the occurring situation as well as steps for generating and executing a course of action to actively collect the missing information. The process of generating the information-gathering solutions was described in terms of sensory actions using sensor-based planning and greedy approaches. The use of planning was moti-

vated by its ability to handle complex situations involving lack of information in an automatic and flexible way.

As there can be situations involving the presence of a high number of hypotheses about how expectations can be checked to hold or not, planning might be costly in terms of computational time and memory resources. For this reason, we have presented an alternative approach that greedily select one piece of information to look for at a time. The information to select is the one that is predicted to reduce the uncertainty in the action outcomes.

Chapter 7 presented a case study of using active information-gathering to respond to plan-execution failures caused by ambiguity in anchoring a symbol to a perceived candidate object. We showed that such failures are characterized by lack of perceptual information about properties of perceived objects. Therefore, the same schema presented in chapter 6 has been successfully applied to devise active information-gathering solutions to recover from ambiguous situations in anchoring.

Limitations and Open Issues

One of the main limitations of semantic knowledge-based execution monitoring is that it is applicable only when domains have descriptive semantic knowledge available, i.e., domains where knowledge about objects can be organized in classes and relations between classes. In addition, there are some open issues that need to be investigated further in order improve the practicability of semantic knowledge-based execution monitoring. These open issues have been identified separately for each chapter of the thesis, and they are exposed in the following paragraphs.

The first issue is related to the process of selecting the objects that need to have their constraints (implicit expectations) checked by the monitoring process against the available perceptual information. In our current implementation, we use an ad hoc procedure that consists in selecting objects that appear in the positive effects stated in the model of the executed action. In crisp *SKEMon*, for instance, the action `(enter r1)` has as positive effect `(robot-in = r1)`, which states that the robot will be in `r1`; therefore `r1` is selected as the object to check by *SKEMon*. This procedure works well when the positive effects refer to a small number of objects. For actions that involve a large number of objects, this might be problematic; thus, this issue needs further research aiming at finding alternative procedures for selecting the few objects that are most important for the monitoring process.

In the current probabilistic *SKEMon* process, number constraints are limited to be over atomic concepts. This limitation was imposed in order to preserve the independence assumption about making observations, i.e., observing one object is independent of observing another. Using the current settings, we cannot have a number constraint about beds and at the same time another number constraint about big-beds because observing an object that is of type bed is

no longer independent of observing an object of type big-bed. Therefore, it is worth investigating efficient ways of computing the posterior of the outcomes of an executed action when there are observation random-variables associated with concepts at different levels of the concept taxonomy. Another issue worth investigation in probabilistic *SKEMon* is addressing the computational complexity of the sensing model, for instance by using approximate inference techniques that are widely studied in the context of inference in Bayesian networks.

Regarding sensor-based planning for dealing with situations of lack of information in crisp *SKEMon*, there are two open issues that need to be addressed. First, situations involving lack of information might be very complex due to a large number of implicit expectations with unknown truth values. Solving such situations by planning can be computationally demanding. A possible solution would be to identify only a subset of expectations to check and then plan to collect information related to them. The second open issue is related to the interaction between the execution of the task plan and the execution of information-gathering actions. In our current implementation, information-gathering plans are not allowed to modify the state reached by the execution of the task plan. This is an extreme constraint that might prevent finding information gathering plans; thus, further research is needed to find ways to coordinate the execution of both plans so that the assigned task is achieved successfully.

Appendix A
Appendix

A.1 Semantic Knowledge Base

A.1.1 Manipulation Domain

```
;-------------------------------------------------------------------------------------------
; Relationships
;-------------------------------------------------------------------------------------------

(defrelation has-handle :domain cup :range handle)
(defrelation has-cover  :domain container :range cover)
(defrelation has-cap    :domain bottle :range cap)

;-------------------------------------------------------------------------------------------
; Atomic Concepts
;-------------------------------------------------------------------------------------------

(defconcept handle)
(defconcept cover)
(defconcept cap)
(defconcept container)

;-------------------------------------------------------------------------------------------
; Defined Concepts
;-------------------------------------------------------------------------------------------

(defconcept cup
   :is (and container (:exactly 1 has-handle)(:exactly 0 has-cover)(:exactly 0 has-cap)))
(defconcept glass
   :is (and container (:exactly 0 has-handle)(:exactly 0 has-cover)(:exactly 0 has-cap)))
(defconcept bottle
   :is (and container (:exactly 0 has-handle)(:exactly 0 has-cover)(:exactly 1 has-cap)))
(defconcept box
   :is (and container (:exactly 0 has-handle)(:exactly 1 has-cover)(:exactly 0 has-cap)))
(defconcept bowl
   :is-primitive (and container (:exactly 0 has-handle)(:exactly 0 has-cover)
                      (:exactly 0 has-cap)))
```

A.1.2 Navigation Domain

```
;-------------------------------------------------------------------------------------------
; Relationships
;-------------------------------------------------------------------------------------------
```

```
(defrelation has-oven :domain kitchen :range  oven)
(defrelation has-bed :domain room :range  bed)
(defrelation has-sofa :domain room :range  sofa)
(defrelation has-table :domain location :range  table)
(defrelation has-tv-set :domain room :range  tv-set)
(defrelation has-sink :domain (or kitchen bathroom utility-room) :range  sink)
(defrelation has-tub :domain (or bathroom utility-room) :range  tub)
(defrelation has-chair :domain location :range  chair)
(defrelation has-fridge :domain room :range  fridge)
(defrelation has-pc :domain room :range  pc)
(defrelation has-clothes-dryer :domain room :range  clothes-dryer)
(defrelation has-washing-machine :domain utility-room :range washing-machine)
(defrelation has-plant :domain location :range  plant)

;-------------------------------------------------------------------------------------------
; Atomic Concepts
;-------------------------------------------------------------------------------------------

;-- Items
(defconcept oven)  (defconcept bed)
(defconcept sofa)  (defconcept table)
(defconcept tv-set)(defconcept sink)
(defconcept tub)   (defconcept chair)
(defconcept fridge)(defconcept washing-machine)
(defconcept plant) (defconcept pc)
(defconcept clothes-dryer)

;-- Locations
(defconcept room )(defconcept corridor)

;-------------------------------------------------------------------------------------------
; Defined Concepts
;-------------------------------------------------------------------------------------------

(defset location :is '(corridor room))
(defconcept  bedroom
   :is (and room (:at-least 1 has-bed)(:at-most 1 has-sofa)
           (:exactly 0 has-sink)(:exactly 0 has-oven)
       (:exactly 0 has-tub)(:exactly 0 has-washing-machine)
       (:exactly 0 has-clothes-dryer)  ))
(defconcept  living-room
    :is (and room (:at-least 1 has-sofa)
            (:exactly  1 has-tv-set)
            (:exactly  0 has-sink)(:exactly  0 has-oven)(:exactly 0 has-tub)
        (:exactly  0 has-washing-machine)(:exactly  0 has-clothes-dryer)))
(defconcept kitchen
   :is (and room (:at-least 1 has-sink)(:exactly 1 has-oven)(:at-least 1 has-fridge)
         (:at-least 1 has-table)(:exactly  0 has-pc)(:at-most 1 has-sofa)
           (:exactly  0 has-bed)(:exactly 0 has-tub)(:exactly 0 has-washing-machine )
       (:exactly  0 has-clothes-dryer) ))
(defconcept bathroom
   :is (and room (:at-least 1 has-sink)(:exactly 1 has-tub)(:at-most 2 has-chair)
         (:at-most 1 has-table)(:exactly 0 has-pc)
           (:exactly  0 has-bed)(:exactly 0 has-sofa)(:exactly 0 has-fridge)
       (:exactly 0 has-oven)(:exactly 0 has-washing-machine )))
(defconcept office
   :is (and room (:at-least 1 has-table)(:at-least 1 has-chair)(:at-least 1 has-pc)
            (:exactly 0 has-bed) (:at-most 1 has-sofa)(:exactly 0 has-fridge)
            (:exactly 0 has-sink)(:exactly 0 has-oven)(:exactly 0 has-tub)
        (:exactly 0 has-washing-machine)(:exactly 0 has-clothes-dryer)))
(defconcept utility-room
   :is (and room   (:at-least 1 has-washing-machine)(:exactly 1 has-clothes-dryer)
          (:exactly 0 has-oven)(:exactly 0 has-bed)(:exactly 0 has-sofa)
        (:exactly 0 has-pc)(:exactly 0 has-fridge)  ))
```

A.2 Actions and Methods for Information Gathering

This appendix lists PTLPLAN planning actions and PC-SHOP methods that were used to generate information-gathering plans for checking number and type constraints (i.e., `:at-least`, `:at-most`, `:exactly`, `:all`, and `:some`) for a navigation domain. Notice that the `:exactly` constraint does not have an action for its own, as its evaluation can be planned for using the actions associated with `:at-least` and `:at-most` constraints.

A.2.1 PTLPLAN Actions

```
;-------------------------------------------------------------------------------------------
;                      (eval-at-least ?n ?r ?x)
; Purpose: evaluates the truth value of the constraint (:at-least ?n ?r) for individual ?x
;-------------------------------------------------------------------------------------------

(ptl-action
 :name    (eval-at-least ?n ?r ?x)
 :precond ( ( (?p)(place ?p)(robot-at = ?p))
            ( (?r)(role  ?r)(and  (not (checked ?r ?p)) (can-check ?r ?p)))
            ( (?x)(room  ?x)(and  (part-of ?p = ?x)(not (nec (at-least ?n ?r ?x))))))


 :results (and (checked ?r ?p = t)
           (cond ((at-least ?n ?r ?x)
                  (obs (at-least ?n ?r ?x = t)) )
                 ((and (at-least ?n ?r ?x = f)
                    (forall(?l)(can-check ?r ?l)(checked ?r ?l)))
                  (obs (at-least ?n ?r ?x = f)))
                 ((true)
                  (and (obs (at-least  ?n ?r ?x = f))
                       (at-least ?n ?r ?x = t f))))  )   )

;-------------------------------------------------------------------------------------------
;                      (eval-at-most ?n ?r ?x)
; Purpose: evaluates the truth value of the constraint (:at-most ?n ?r) for individual ?x
;-------------------------------------------------------------------------------------------

(ptl-action
 :name    (eval-at-most ?n ?r ?x)
 :precond ( ( (?p)(place ?p)(robot-at = ?p))
            ( (?r)(role  ?r)(and  (can-check ?r ?p)(not (checked ?r ?p)) ))
            ( (?x)(room  ?x)(and  (part-of ?p = ?x)(not (nec (at-most ?n ?r ?x))))))

 :results (and (checked ?r ?p = t)
           (cond ((at-most ?n ?r ?x = f)
                  (obs (at-most ?n ?r ?x = f)) )
                 ((and (at-most ?n ?r ?x = t)
                    (forall(?l)(can-check ?r ?l)(checked ?r ?l)))
                  (obs (at-most ?n ?r ?x = t)))
                 ((true)
                  (and (obs (at-most  ?n ?r ?x = t))
                       (at-most ?n ?r ?x = t f))))  )   )
```

```
;-------------------------------------------------------------------------------------------
;                      (eval-all ?r ?c ?x)
; Purpose: evaluates the truth value of the constraint (:all ?r ?c) for individual ?x
;-------------------------------------------------------------------------------------------

(ptl-action
 :name    (eval-all ?r ?c ?x)
 :precond ( ( (?p)(place ?p)(robot-at = ?p))
            ( (?r)(role  ?r)(and  (can-check ?r ?p)(not (checked ?r ?p)) ))
            ( (?x)(room  ?x)(and  (part-of ?p = ?x)(not (nec (all ?r ?c ?x))))))

 :results (and (checked ?r ?p = t)
           (cond ((all ?r ?c ?x = f)
                  (obs (all ?r ?c ?x = f)) )
                 ((and (all ?r ?c ?x = t)
                    (forall(?l)(can-check ?r ?l)(checked ?r ?l)))
                  (obs (all ?r ?c ?x = t)))
                 ((true)
                  (and (obs (all  ?r ?c ?x = t))
                       (all ?r ?c ?x = t f))))  )   )

;-------------------------------------------------------------------------------------------
;                      (eval-some ?r ?c ?x)
; Purpose: evaluates the truth value of the constraint (:some ?r ?c) for individual ?x
;-------------------------------------------------------------------------------------------

(ptl-action
 :name    (eval-some ?r ?c ?x)
 :precond ( ( (?p)(place ?p)(robot-at = ?p))
            ( (?r)(role  ?r)(and  (can-check ?r ?p)(not (checked ?r ?p)) ))
            ( (?x)(room  ?x)(and  (part-of ?p = ?x)(not (nec (some ?r ?c ?x))))))

 :results (and (checked ?r ?p = t)
           (cond ((some ?r ?c ?x = t)
                  (obs (some ?r ?c ?x = t)) )
                 ((and (some ?r ?c ?x = f)
                    (forall(?l)(can-check ?r ?l)(checked ?r ?l)))
                  (obs (some ?r ?c ?x = f)))
                 ((true)
                  (and (obs (some  ?r ?c ?x = f))
                       (some ?r ?c ?x = t f))))  )   )
```

A.2.2 PC-SHOP Methods

```
;-------------------------------------------------------------------------------------------
;                      (!check-at-least ?n ?r ?x)
; Purpose: generates a plan to evaluate the truth value of the constraint (:at-least ?n ?r)
;          for individual ?x
;-------------------------------------------------------------------------------------------

(method (!check-at-least ?n ?r ?x)

      /* Alternative 1*/
     (((?p)(place ?p)
       (and (finish = f)(robot-at = ?p)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (:immediate eval-at-least ?n ?r ?x)
               (:cond ((at-least ?n ?r ?x)
                       (:immediate !eval-termination))
                      ((not (at-least ?n ?r ?x))
                       (!check-at-least ?n ?r ?x))))
```

```
     /* Alternative 2*/
     (((?p)(place ?p)(and (finish = f)(can-check ?r ?p)
                          (not (checked ?r ?p)))))
     (:ordered (move ?p)(!check-at-least ?n ?r ?x))

     /* Alternative 3*/
     (true)
     (!eval-termination) )

;-------------------------------------------------------------------------------------------
;                      (!check-at-most ?n ?r ?x)
; Purpose: generates a plan to evaluate the truth value of the constraint (:at-most ?n ?r)
;          for individual ?x
;-------------------------------------------------------------------------------------------

(method (!check-at-most ?n ?r ?x)

      /* Alternative 1*/
     (((?p)(place ?p)
       (and (finish = f)(robot-at = ?p)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (:immediate eval-at-most ?n ?r ?x)
               (:cond ((at-most ?n ?r ?x = f)
                       (:immediate !eval-termination))
                      ((at-most ?n ?r ?x = t)
                       (!check-at-most ?n ?r ?x))))

     /* Alternative 2*/
     (((?p)(place ?p)(and (finish = f)(can-check ?r ?p)
                          (not (checked ?r ?p)))))
     (:ordered (move ?p)(!check-at-most ?n ?r ?x))

     /* Alternative 3*/
     (true)
     (!eval-termination) )

;-------------------------------------------------------------------------------------------
;                      (!check-all ?r ?c ?x)
; Purpose: generates a plan to evaluate the truth value of the constraint (:all ?r ?c)
;          for individual ?x
;-------------------------------------------------------------------------------------------

(method (!check-all ?r ?c ?x)

      /* Alternative 1*/
     (((?p)(place ?p)
       (and (finish = f)(robot-at = ?p)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (:immediate eval-all ?r ?c ?x)
               (:cond ((all ?r ?c ?x = f)
                       (:immediate !eval-termination))
                      ((all ?r ?c ?x)
                       (!check-all ?r ?c ?x))))

     /* Alternative 2*/
     (((?p)(place ?p)(and (finish = f)(can-check ?r ?p)
                          (not (checked ?r ?p)))))
     (:ordered (move ?p)(!check-all ?r ?c ?x))

     /* Alternative 3*/
     (true)
     (!eval-termination) )
```

```
;-------------------------------------------------------------------------------------------
;                     (!check-some ?r ?c ?x)
; Purpose: generates a plan to evaluate the truth value of the constraint (:some ?r ?c)
;          for individual ?x
;-------------------------------------------------------------------------------------------

(method (!check-some ?r ?c ?x)

      /* Alternative 1*/
     (((?p)(place ?p)
       (and (finish = f)(robot-at = ?p)(can-check ?r ?p)
            (not (checked ?r ?p)))))
     (:ordered (:immediate eval-all ?r ?c ?x)
               (:cond ((some ?r ?c ?x = t)
                       (:immediate !eval-termination))
                      ((some ?r ?c ?x = f)
                       (!check-some ?r ?c ?x))))

     /* Alternative 2*/
     (((?p)(place ?p)(and (finish = f)(can-check ?r ?p)
                     (not (checked ?r ?p)))))
     (:ordered (move ?p)(!check-some ?r ?c ?x))

     /* Alternative 3*/
     (true)
     (!eval-termination) )

;-------------------------------------------------------------------------------------------
;                     (!eval-termination)
; Purpose: checks whether planning should be terminated depending on the truth value
;          of planning-formula
;-------------------------------------------------------------------------------------------

(method (!eval-termination)
      /* Alternative 1*/
     ((()(nec planning-formula)))
      (stop success)
      /* Alternative 2*/
      ((()(nec (not planning-formula))))
      (stop violated)
     /* Alternative 3*/
     (true)
     (:nop))
```

Bibliography

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *Int. Journal of Robotics Research*, 17(4):315–337, 1998.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[4] N. Balac, D. M. Gaines, and D. Fisher. Using regression trees to learn action models. In *IEEE Int. Conf. on Systems, Man and Cybernetics*, pages 3378–3383, 2000.

[5] M. Beetz. Structured reactive controllers. In *3rd Int. Conf. on Autonomous Agents*, pages 228–235, 1999.

[6] M. Beetz. Runtime plan adaptation in structured reactive controllers. In *4th Int. Conf. on Autonomous Agents*, pages 19–20, 2000.

[7] M. Beetz and D. McDermott. Fast probabilistic plan debugging. In *European Conf. on Planning*, pages 77–90, 1997.

[8] S. Benson. Inductive learning of reactive action models. In *Int. Conf. on Machine Learning*, pages 47–54, 1995.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May, 2001.

[10] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *17th Int. Joint Conf. on Artificial Intelligence*, pages 473–478, 2001.

[11] M. Bjäreland. *Model-based Execution Monitoring*. PhD thesis, Linköping University, Sweden, 2001.

[12] M. Bjäreland and G. Fodor. Ontological control. In *9th Int. Workshop on Principles of Diagnosis*, 1998.

[13] M. R. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda. Mars polar lander fault identification using model-based testing. In *8th Int. Conf. on Engineering of Complex Computer Systems*, pages 163–170, 2002.

[14] J. Blythe. Planning with external events. In *10th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 94–101, 1994.

[15] J. Blythe. *Planning under Uncertainty in Dynamic Domains*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1998.

[16] J. Blythe. Decision-theoretic planning. *AI Magazine*, 20(2):37–54, 1999.

[17] B. Bonet and H. Geffner. GPT: A tool for planning with uncertainty and partial information. In *IJCAI Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87, 2001.

[18] J. C. Bongard and H. Lipson. Automated damage diagnosis and recovery for remote robotics. In *IEEE Int. Conf. on Robotics and Automation*, pages 3545–3550, 2004.

[19] J. C. Bongard and H. Lipson. Automated robot function recovery after unanticipated failure or environmental change using a minimum of hardware trials. In *NASA/DoD Conf. on Evolvable Hardware*, pages 169– 176, 2004.

[20] A. Bouguerra and L. Karlsson. Hierarchical task planning under uncertainty. In *3rd Italian Workshop on Planning and Scheduling*, 2004.

[21] A. Bouguerra and L. Karlsson. Pc-shop: A probabilistic-conditional hierarchical task planner. *Intelligenza Artificiale*, 2(4):44–50, 2005.

[22] A. Bouguerra, L. Karlsson, and A. Saffiotti. Situation assessment for sensor-based recovery planning. In *17th European Conf. on AI*, pages 673–677, 2006.

[23] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[24] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[25] J. Bresina, D. Richard, M. Nicolas, R. Sailesh, S. David, and W. Rich. Planning under continuous time and resource uncertainty: A challenge for AI. In *18th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 77–84, 2002.

[26] M. Broxvall, S. Coradeschi, L. Karlsson, and A. Saffiotti. Have another look: On failures and recovery planning in perceptual anchoring. In *ECAI Workshop on Cognitive Robotics*, 2004.

[27] M. Broxvall, S. Coradeschi, L. Karlsson, and A. Saffiotti. Recovery planning for ambiguous cases in perceptual anchoring. In *20th National Conf. on Artificial Intelligence*, pages 1254–1260, 2005.

[28] W. Burgard, D. Fox, and S. Thrun. Active mobile robot localization by entropy minimization. In *2nd Euromicro Workshop on Advanced Mobile Robots*, pages 155–162, 1997.

[29] J. Burlet, O. Aycard, and T. Fraichard. Robust motion planning using markov decision processes and quadtree decomposition. In *IEEE Int. Conf. on Robotics and Automation*, pages 2820–2825, 2004.

[30] J. Carlson and R. R. Murphy. Reliability analysis of mobile robots. In *IEEE Int. Conf. on Robotics and Automation*, pages 274–281, 2003.

[31] J. Carlson, R. R Murphy, and A. Nelson. Follow-up analysis of mobile robot failures. In *IEEE Int. Conf. on Robotics and Automation*, pages 4987–4994, 2004.

[32] A. Chella, M. Cossentino, R. Pirrone, and A. Ruisi. Modeling ontologies for robotic environments. In *14th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 77–80, 2002.

[33] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *5th Int. Conf. on Artificial Intelligence Planning and Scheduling*, pages 300–307, 2000.

[34] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[35] S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: preliminary report. In *17th National Conf. on Artificial Intelligence*, pages 129–135, 2000.

[36] S. Coradeschi and A. Saffiotti. Perceptual anchoring of symbols for action. In *17th Int. Joint Conf. on Artificial Intelligence*, pages 407–412, 2001.

[37] S. Coradeschi and A. Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, 2003. Special issue on perceptual anchoring.

[38] K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[39] R. Dearden, F. Huttner, R. Simmons, V. Verma, S. Thrun, and T. Willeke. Real-time fault detection and situational awareness for rovers: Report on the mars technology program task. In *IEEE Aerospace Conf.*, 2004.

[40] R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith, and R. Washington. Incremental contingency planning. In *ICAPS Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

[41] G. DeGiacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *6th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 453–465, 1998.

[42] O. Despouys and F. F. Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *5th European Conf. on Planning*, pages 278–293, 2000.

[43] M. B. Dias, S. Lemai, and N. Muscettola. A real-time rover executive based on model-based reactive planning. In *7th Int. Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2003.

[44] S. Ekvall, D. Kragic, and P. Jensfelt. Object detection and mapping for service robot tasks. *Robotica*, 25(2):175–187, 2007.

[45] E. A. Emerson. *Temporal and modal logic*, pages 995–1072. MIT Press, 1990.

[46] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[47] J. L. Fernández, R. Sanz, and A. R. Diéguez. Probabilistic models for monitoring and fault diagnosis: Application and evaluation in a mobile robot. *Applied Artificial Intelligence*, 18(1):43–67, 2004.

[48] J. L. Fernández and R. G. Simmons. Robust execution monitoring for navigation plans. In *IEEE/RSJ Conf. on Intelligent Robots and Systems*, pages 551–557, 1998.

[49] C. Ferrel. Failure recognition and fault tolerance of an autonomous robot. *Adaptive Behaviour*, 2(4):375–398, 1994.

[50] M. Fichtner, A. Großmann, and M. Thielscher. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57(2-4):371–392, 2003.

[51] R. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[52] R. E. Fikes. Monitored execution of robot plans produced by STRIPS. Technical Report 55, AI Center, SRI International, April 1971.

[53] R. E. Fikes, P. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[54] R. J. Firby. Building symbolic primitives with continuous control routines. In *1st Int. Conf. on Artificial intelligence planning systems*, pages 62–69, 1992.

[55] D. Fox, W. Burgard, and S. Thrun. Active markov localization for mobile robots. *Robotics and Autonomous Systems*, 25:195–207, 1998.

[56] D. Fox, S. Thrun, F. Dellaert, and W. Burgard. Particle filters for mobile robot localization. In A. Doucet, N. de Freitas, and N. Gordon, editors, *Sequential Monte Carlo Methods in Practice*. Springer Verlag, 2000.

[57] G. Fraser, G. Steinbauer, and F. Wotawa. Plan execution in dynamic environments. In *18th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 208–217, 2005.

[58] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *16th Int. Joint Conf. on Artificial Intelligence*, pages 1300–1309, 1999.

[59] T. Fujii and T. Ura. Development of an autonomous underwater robot "twin-burger" for testing intelligent behaviors in realistic environments. *Autonomous Robots*, 3(2-3):285–296, 1996.

[60] C. Galindo, A. Saffiotti, S. Coradeschi, P. Buschka, J.A. Fernández-Madrigal, and J. González. Multi-hierarchical semantic maps for mobile robotics. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pages 3492–3497, 2005.

[61] J. Gancet and S. Lacroix. PG2P: A perception-guided path planning approach for long range autonomous navigation in unknown natural environments. In *IEEE/RSJ Int. Conf. of Intelligent Robots and Systems*, pages 2992– 2997, 2003.

[62] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *10th National Conf. on Artificial Intelligence*, pages 809–815, 1992.

[63] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning, Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[64] M. L. Ginsberg. Universal planning: an (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.

[65] A. Goel, E. Stroulia, Z. Chen, and P. Rowland. Model-based reconfiguration of schema-based reactive control architectures. In *15th Int. Joint Conf. on Artificial Intelligence*, 1997.

[66] D. Gu and H. Hu. Teaching robots to plan through Q-learning. *Robotica*, 23(2):139–147, 2005.

[67] Y. Guo, A. Qasem, and J. Heflin. Large scale knowledge base systems: An empirical evaluation perspective. In *National Conf. on AI*, 2006.

[68] G. Hager and M. Mintz. Computational methods for task-directed sensor data fusion and sensor planning. *Int. Journal of Robotics Research*, 10(4):285–313, 1991.

[69] K. Z. Haigh and M. M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In *1st Int. Conf. on Autonomous Agents*, pages 363–370, 1997.

[70] K. Z. Haigh and M. M. Veloso. Learning situation-dependent costs: Improving planning from probabilistic robot execution. In *2nd Int. Conf. on Autonomous Agents*, pages 231–238, 1998.

[71] K. Z. Haigh and M. M. Veloso. Planning, execution and learning in a robotic agent. In *4th Int. Conf. on Artificial Intelligence Planning Systems*, pages 120–127, 1998.

[72] J. Hertzberg and A. Saffiotti, editors. *Workshop on Semantic Information in Robotics, IEEE Int. Conf. on Robotics and Automation*, April 2007.

[73] J. Hois, M. Wünstel, J. A. Bateman, and T. Röfer. Dialog-based 3D-image recognition using a domain ontology. In *Int. Conf. Spatial Cognition*, pages 107–126, 2006.

[74] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents – summary of an agent infrastructure. In *Int. Conf. on Autonomous Agents*, 2001.

[75] A. E. Howe. Analyzing failure recovery to improve planner design. In *10th National Conf. on Artificial Intelligence*, pages 387–392, 1992.

[76] A. E. Howe. Improving the reliability of artificial intelligence planning systems by analyzing their failure recovery. *IEEE Trans. on Knowledge and Data Engineering*, 7(1):14–25, 1995.

[77] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE Int. Conf. on Robotics and Automation*, pages 43–49, 1996.

[78] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992.

[79] M. Jaeger. Relational Bayesian networks. In *13th Conf. on Uncertainty in Artificial Intelligence*, pages 266–273, 1997.

[80] F. Kabanza and K. BenLamine. Specifying failure and progress conditions in a behavior-based robot programming system. In *4th Int. Cognitive Robotics Workshop*, 2004.

[81] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[82] L. Karlsson. Conditional progressive planning under uncertainty. In *17th Int. Joint Conf. on Artificial Intelligence*, pages 431–438, 2001.

[83] L. Karlsson, A. Bouguerra, M. Broxvall, S. Coradeschi, and A. Saffiotti. To secure an anchor- a recovery planning approach to ambiguity in perceptual anchoring. *AI Communications*, 21(1):1–14, 2008.

[84] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, volume 3, pages 2149 – 2154, 2004.

[85] S. Koenig and Y. Liu. Sensor planning with non-linear utility functions. In *European Conf. on Planning*, pages 265–277, 2000.

[86] D. Koller, A. Y. Levy, and A. Pfeffer. P-classic: A tractable probabilistic description logic. In *AAAI Conf.*, pages 390–397, 1997.

[87] S. Kovacic, A. Leonardis, and F. Pernus. Planning sequences of views for 3-D object recognition and pose determination. *Pattern Recognition*, 31:1407–1417, 1998.

[88] K. B. Lamine and F. Kabanza. History checking of temporal fuzzy logic formulas for monitoring behavior-based mobile robots. In *12th IEEE Int. Conf. on Tools with Artificial Intelligence*, pages 312–319, 2000.

[89] S. Lemai and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *19th National Conf. on Artificial Intelligence*, pages 617–622, 2004.

[90] M. L. Littman and S. M. Majercik. Large-scale planning under uncertainty: A survey. In *Workshop on Planning and Scheduling for Space*, 1997.

[91] A. Loutfi, S. Coradeschi, L. Karlsson, and M. Broxvall. Putting olfaction into action: Using an electronic nose on an multi-sensing mobile robot. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 337–342, 2004.

[92] D. Lowe. Distinctive image features from scale-invariant keypoints. *Int. Journal of Computer Vision*, 60(2):91–110, 2004.

[93] T. Lueth and T. Laengle. Fault-tolerance and error recovery in an autonomous robot with distributed controlled components. In *2nd IEEE Int. Symposium on Distributed Autonomous Robotic Systems*, 1994.

[94] T. Lukasiewicz. Probabilistic description logic programs. *Int. Journal of Approximate Reasoning*, 45(2):288–307, 2007.

[95] T. Lukasiewicz and U. Straccia. Description logic programs under probabilistic uncertainty and fuzzy vagueness. In *9th European Conf. on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 187–198, 2007.

[96] B. Lussier, R. Chatila, F. Ingrand, M.O. Killijian, and D. Powell. On fault tolerance and robustness in autonomous systems. In *3rd IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2004.

[97] R. MacGregor. Retrospective on loom. Technical report, Information Sciences Institute, University of Southern California, 1999.

[98] R. G. Martínez and D. Borrajo. An integrated approach of learning, planning, and execution. *Journal of Intelligent Robotics Systems*, 29(1):47–78, 2000.

[99] C. E. McCarthy and M. E. Pollack. Towards focused plan monitoring: A technique and an application to mobile robots. *Autonomous Robots*, 9(1):71–81, 2000.

[100] K. Mikolajczyk, B. Leibe, and B. Schiele. Local features for object class recognition. In *10th IEEE Int. Conf. on Computer Vision*, pages 1792–1799, 2005.

[101] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *19th Int. Joint Conf. on Artificial Intelligence*, pages 1352–1359, 2005.

[102] J. Miura and Y. Shirai. Vision and motion planning for a mobile robot under uncertainty. *Int. Journal of Robotics Research*, 16(6):806–825, 1997.

[103] R. Möller, V. Haarslev, and M. Wessel. On the scalability of description logic instance retrieval. In *Int. Workshop on Description Logics*, 2006.

[104] R. R. Murphy and D. Hershberger. Handling sensing failures in autonomous mobile robots. *Int. Journal of Robotics Research*, 18(4):382 – 400, 1999.

[105] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[106] K. L. Myers. A procedural knowledge approach to task-level control. In *3rd Int. Conf. on Artificial Intelligence Planning Systems*, pages 158–165, 1996.

[107] D. Nau, Y. Cao, A. Lotem, and H. Muñoz Avila. SHOP: Simple hierarchical ordered planner. In *16th Int. Joint Conf. on Artificial Intelligence*, pages 968–973, 1999.

[108] D. Nau, Y. Cao, A. Lotem, and H. Muñoz Avila. SHOP and M-SHOP: Planning with ordered task decomposition. Technical Report CS-TR-4157, University of Maryland, College Park, MD, 2000.

[109] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Artificial Intelligence Research*, 20:379–404, 2003.

[110] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454, 1995.

[111] A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. S. Kim. CLARAty: An architecture for reusable robotic software. In *SPIE Aerosense Conf.*, 2003.

[112] B. Neumann and R. Möller. On scene interpretation with description logics. *Image and Vision Computing*, 26(1):82–101, 2008.

[113] I. Nourbakhsh and M. Genesereth. Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots Journal*, 3(1):49–67, 1996.

[114] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: An office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[115] I. R. Nourbakhsh, A. Soto, J. Bobenage, S. Grange, R. Meyer, and R. Lutz. An effective mobile robot educator with a full-time job. *Artificial Intelligence*, 114(1-2):95–124, 1999.

[116] A. Nüchter, O. Wulf, K. Lingemann, J. Hertzberg, B. Wagner, and H. Surmann. 3D mapping with semantic knowledge. In *RoboCup Int. Symposium*, pages 335–346, 2005.

[117] N. Onder and M. E. Pollack. Contingency selection in plan generation. In *4th European Conf. on Planning*, pages 364–376, 1997.

[118] A. Efros P. E. Rybski. Overview of the 2007 semantic robot vision challenge competition. In *AAAI Mobile Robot Competition and Exhibition Workshop, National Conf. on Artificial Intelligence*, 2007.

[119] H. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning probabilistic relational planning rules. In *Int. Conf. on Automated Planning and Scheduling*, pages 73–82, 2004.

[120] B. Pell, E. B. Gamble, E. Gat, R. Keesing, J. Kurien, W. Millar, P. P. Nayak, C. Plaunt, and B. C. Williams. A hybrid procedural/deductive executive for autonomous spacecraft. In *2nd Int. Conf. on Autonomous Agents*, pages 369–376, 1998.

[121] O. Pettersson, L. Karlsson, and A. Saffiotti. Model-free execution monitoring in behavior-based robotics. *IEEE Trans. on Systems, Man and Cybernetics, Part B*, 37(4):890–901, 2007.

[122] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[123] P. Pirjanian. Reliable reaction. In *IEEE Int. Conf. on Multisensor Fusion and Integration for Intelligent Systems*, pages 158–165, 1996.

[124] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, 9 2003.

[125] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.

[126] F. Py and F. Ingrand. Dependable execution control for autonomous robots. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, volume 2, pages 1136–1141, 2004.

[127] A. Ramisa, S. Vasudevan, D. Scaramuzza, R. de Mántaras, and R. Siegwart. A tale of two object recognition methods for mobile robots. In *Int. Conf. on Computer Vision Systems*, pages 353–362, 2008.

[128] N. Roy, G. Gordon, and S. Thrun. Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research*, 23:1–40, 2005.

[129] T. Russ, R. MacGregor, B. Salemi, K. Price, and R. Nevatia. VEIL: Combining semantic knowledge with image understanding. In *ARPA Image Understanding Workshop*, 1996.

[130] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 14–15. Prentice Hall, second edition, 2003.

[131] A. Saffiotti. *Autonomous Robot Navigation: a fuzzy logic approach*. PhD thesis, Faculté de Sciences Appliqueés, Université Libre de Bruxelles, 1998.

[132] A. Saffiotti and M. Broxvall. PEIS ecologies: Ambient intelligence meets autonomous robotics. In *Int. Conf. on Smart Objects and Ambient Intelligence*, pages 275–280, 2005.

[133] A. Saffiotti, K. Konolige, and E. H. Ruspini. A multivalued logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

[134] A. Saffiotti, E. H. Ruspini, and K. Konolige. Using fuzzy logic for mobile robot control. In H. Prade, D. Dubois, and H. J. Zimmermann, editors, *International Handbook of Fuzzy Sets and Possibility Theory*, volume 5. Kluwer Academic Publishers Group, 1997.

[135] M. D. Schmill, T. Oates, and P. R. Cohen. Learning planning operators in real-world, partially observable environments. In *Artificial Intelligence Planning Systems*, pages 246–253, 2000.

[136] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *10th Int. Joint Conf. on Artificial Intelligence*, pages 1039–1046, 1987.

[137] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[138] R. Simmons. Structured control for autonomous robots. *IEEE Trans. on Robotics and Automation*, 10(1):34–43, 1994.

[139] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Int. Joint Conf. on Artificial Intelligence*, pages 1080 – 1087, 1995.

[140] M. A. Sipe and D. Casasent. Global feature space neural network for active object recognition. In *Int. Joint Conf. on Neural Networks*, pages 3128–3133, 1999.

[141] A. Tate. Generating project networks. In *5th Int. Joint Conf. on Artificial Intelligence*, pages 888–93, 1977.

[142] C. Theobalt, J. Bos, T. Chapman, A. Espinosa-Romero, M. Fraser, G. Hayes, E. Klein, T. Oka, and R. Reeve. Talking to Godot: Dialogue with a mobile robot. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pages 1338–1343, 2002.

[143] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. R. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second-generation museum tour-guide robot. In *Int. Conf. on Robotics and Automation*, pages 1999–2005, 1999.

[144] R. M. Turner, P. S. Eaton, and M. J. Dempsey. Handling unanticipated events in single and multiple AUV systems. In *IEEE Oceanic Engineering Society Conf.*, 1994.

[145] P. Varakantham, R. T. Maheswaran, T. Gupta, and M. Tambe. Towards efficient computation of error bounded solutions in POMDPs: Expected value approximation and dynamic disjunctive beliefs. In *20th Int. Joint Conf. on Artificial Intelligence*, pages 2638–2644, 2007.

[146] M. M. Veloso, M. E. Pollack, and M.T. Cox. Rationale-based monitoring for planning in dynamic environments. In *4th Int. Conf. on Artificial Intelligence Planning Systems*, pages 171–179, 1998.

[147] V. Verma, J. Fernandez, and R. Simmons. Probabilistic models for monitoring and fault diagnosis. In *2nd IARP and IEEE/RAS Joint Workshop on Technical Challenges for Dependable Robots in Human Environments.*, 2002.

[148] V. Verma, G. Gordon, and R. Simmons. Efficient monitoring for planetary rovers. In *Int. Symposium on Artificial Intelligence and Robotics in Space*, 2003.

[149] V. Verma, G. Gordon, R. Simmons, and S. Thrun. Particle filters for rover fault diagnosis. *IEEE Robotics & Automation Magazine special issue on Human Centered Robotics and Dependability*, 2004.

[150] X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Int. Conf. on Machine Learning*, pages 549–557, 1995.

[151] D. E. Wilkins. *Practical Planning: extending the classical AI paradigm*. Morgan Kaufmann, 1988.

[152] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

[153] D. E. Wilkins, K. L. Myers, and L. P. Wesley. Cypress: planning and reacting under uncertainty. In *ARPA/Rome Laboratory Planning and Scheduling Initiative Workshop*, pages 111–120, 1994.

Publikationer *i serien* Örebro Studies in Technology

1. Bergsten, Pontus (2001) *Observers and Controllers for Takagi – Sugeno Fuzzy Systems*. Doctoral Dissertation.
2. Iliev, Boyko (2002) *Minimum-time Sliding Mode Control of Robot Manipulators*. Licentiate Thesis.
3. Spännar, Jan (2002) *Grey box modelling for temperature estimation*. Licentiate Thesis.
4. Persson, Martin (2002) *A simulation environment for visual servoing*. Licentiate Thesis.
5. Boustedt, Katarina (2002) *Flip Chip for High Volume and Low Cost – Materials and Production Technology*. Licentiate Thesis.
6. Biel, Lena (2002) *Modeling of Perceptual Systems – A Sensor Fusion Model with Active Perception*. Licentiate Thesis.
7. Otterskog, Magnus (2002) *Produktionstest av mobiltelefonantenner i mod-växlande kammare*. Licentiate Thesis.
8. Tolt, Gustav (2003) *Fuzzy-Similarity-Based Low-level Image Processing*. Licentiate Thesis.
9. Loutfi, Amy (2003) *Communicating Perceptions: Grounding Symbols to Artificial Olfactory Signals*. Licentiate Thesis.
10. Iliev, Boyko (2004) *Minimum-time Sliding Mode Control of Robot Manipulators*. Doctoral Dissertation.
11. Pettersson, Ola (2004) *Model-Free Execution Monitoring in Behavior-Based Mobile Robotics*. Doctoral Dissertation.
12. Överstam, Henrik (2004) *The Interdependence of Plastic Behaviour and Final Properties of Steel Wire, Analysed by the Finite Element Metod*. Doctoral Dissertation.
13. Jennergren, Lars (2004) *Flexible Assembly of Ready-to-eat Meals*. Licentiate Thesis.
14. Jun, Li (2004) *Towards Online Learning of Reactive Behaviors in Mobile Robotics*. Licentiate Thesis.
15. Lindquist, Malin (2004) *Electronic Tongue for Water Quality Assessment*. Licentiate Thesis.
16. Wasik, Zbigniew (2005) *A Behavior-Based Control System for Mobile Manipulation*. Doctoral Dissertation.

17. Berntsson, Tomas (2005) *Replacement of Lead Baths with Environment Friendly Alternative Heat Treatment Processes in Steel Wire Production.* Licentiate Thesis.

18. Tolt, Gustav (2005) *Fuzzy Similarity-based Image Processing.* Doctoral Dissertation.

19. Munkevik, Per (2005) *"Artificial sensory evaluation – appearance-based analysis of ready meals".* Licentiate Thesis.

20. Buschka, Pär (2005) *An Investigation of Hybrid Maps for Mobile Robots.* Doctoral Dissertation.

21. Loutfi, Amy (2006) *Odour Recognition using Electronic Noses in Robotic and Intelligent Systems.* Doctoral Dissertation.

22. Gillström, Peter (2006) *Alternatives to Pickling; Preparation of Carbon and Low Alloyed Steel Wire Rod.* Doctoral Dissertation.

23. Li, Jun (2006) *Learning Reactive Behaviors with Constructive Neural Networks in Mobile Robotics.* Doctoral Dissertation.

24. Otterskog, Magnus (2006) *Propagation Environment Modeling Using Scattered Field Chamber.* Doctoral Dissertation.

25. Lindquist, Malin (2007) *Electronic Tongue for Water Quality Assessment.* Doctoral Dissertation.

26. Cielniak, Grzegorz (2007) *People Tracking by Mobile Robots using Thermal and Colour Vision.* Doctoral Dissertation.

27. Boustedt, Katarina (2007) *Flip Chip for High Frequency Applications – Materials Aspects.* Doctoral Dissertation.

28. Soron, Mikael (2007) *Robot System for Flexible 3D Friction Stir Welding.* Doctoral Dissertation.

29. Larsson, Sören (2008) *An industrial robot as carrier of a laser profile scanner. – Motion control, data capturing and path planning.* Doctoral Dissertation.

30. Persson, Martin (2008) *Semantic Mapping Using Virtual Sensors and Fusion of Aerial Images with Sensor Data from a Ground Vehicle.* Doctoral Dissertation.

31. Andreasson, Henrik (2008) *Local Visual Feature based Localisation and Mapping by Mobile Robots.* Doctoral Dissertation.

32. Bouguerra, Abdelbaki (2008) *Robust Execution of Robot Task-Plans: A Knowledge-based Approach.* Doctoral Dissertation.